Application of Internet technologies to Customer Support Services

N J Ramsay 1997 – 1999

11. Errata

Page 30: Replace “Define a simple constructor that this class in the RMI Registry” with “Define a simple constructor that initialises this class in the RMI Registry.”

Page 37: Delete paragraph two of section 5.3.4

Table of contents

1. INTRODUCTION 5

2. BACKGROUND 6

2.1 IBM NEW ZEALAND 6
2.2 INTEGRATED CUSTOMER MANAGEMENT SYSTEM 6
2.3 INTERNET TECHNOLOGIES 7

3. NON-FUNCTIONAL REQUIREMENTS 8

3.1 PERFORMANCE 8
3.1.1 Client Performance 8
3.1.2 Average time to display 8
3.2 EASE OF USE 13
3.2.1 As simple as possible 13
3.2.2 Other advantages of a simple design include 13
3.3 CONFIGURABILITY 13

4. ARCHITECTURAL ALTERNATIVES 15

4.1 HYPERTEXT BASED SOLUTIONS 15
4.1.1 Client Architecture 15
5.1.1 Server Architecture 21
5.1.2 Examples 25
5.2 JAVA BASED FRONTEND 26
5.2.1 Two-tier model 27
5.2.2 Three-tier model 28
5.2.3 Communication 30

6. TOLL ESTIMATOR 33

6.1 BACKGROUND 33
6.2 INTENDED USE 33
6.3 SYSTEM ARCHITECTURE 34
6.3.1 Options 34
6.3.2 Minimising the startup time 34
6.3.3 Java security restrictions 36
6.3.4 Architecture model 37
6.4 CLIENT ARCHITECTURE 37
6.4.1 Startup component 37
6.4.2 User Interface component 38
6.4.3 Messaging component 39
6.4.4 Communication between components 40
6.5 MESSAGE PASSING 41
6.6 MIDDLE TIER ARCHITECTURE 42
6.7 SERVER ARCHITECTURE 42
6.8 FUTURE EXTENSIONS 43
6.8.1 Data source 43
6.8.2 Create Java Beans 43

7. FIND CUSTOMER USE CASE 45

7.1 OPTIONS FOR THE COMMUNICATIONS ARCHITECTURE 45
7.1.1 Requirements 45
7.1.2 Three options 45
7.1.3 Sockets 46
7.1.4 Java RMI and Java JNI based middleware 46
7.1.5 CORBA based middleware 47
7.2 BUILDING THE COMMUNICATIONS ARCHITECTURE 48
7.2.1 Supplier 48
7.2.2 Defining the interface 48

7.2.3 *Compiling an IDL file* 53
7.2.4 *Issues that were addressed* 53
7.3 JAVA CLIENT 54
7.3.1 *Requirements* 54
7.3.2 *Implementation* 54
7.3.3 *Client Usage* 58
7.4 C++ SERVER 59
7.4.1 *CORBA Integration* 60
7.5 PERFORMANCE 60
7.5.1 *CORBA* 60
7.5.2 *Java* 60
7.6 FUTURE ENHANCEMENTS 61
7.6.1 *Split reassessment* 61
7.6.2 *Middle tier* 61
7.6.3 *Data structures* 61
7.6.4 *CORBA services* 62
7.7 CONCLUSIONS 62

8. CONCLUSIONS 63

8.1 USING A CURRENT DATA SET 63
8.2 USING A CURRENT APPLICATION 63
8.3 THE USE OF INTERNET BASED TECHNOLOGIES 63

9. APPENDIX A – TOLL ESTIMATOR DOCUMENTATION 65

9.1 PACKAGE COM.IBM.VOYAGER 65
9.1.1 *Interface COM.ibm.voyager.Callable* 65
9.1.2 *Interface COM.ibm.voyager.Database* 67
9.1.3 *Class COM.ibm.voyager.CallCost* 69
9.1.4 *Class COM.ibm.voyager.CallInfo* 72
9.1.5 *Class COM.ibm.voyager.Message* 78
9.1.6 *Class COM.ibm.voyager.Messenger* 81
9.1.7 *Class COM.ibm.voyager.TollEstimator* 85
9.1.8 *Class COM.ibm.voyager.TollFrame* 89

10. APPENDIX B – FIND CUSTOMER DOCUMENTATION OVERVIEW 93

10.1 PACKAGE GUI12 93
10.1.1 *Interfaces* 93
10.1.2 *Classes* 93
10.2 PACKAGE FC12 93
10.2.1 *Interfacess* 93
10.2.2 *Classes* 93
10.3 PACKAGE POC12 94
10.3.1 *Interfaces* 94
10.3.2 *Classes* 94

11. REFERENCES 95

Table of figures

- FIGURE 1. TWO-TIERED ARCHITECTURE....27
- FIGURE 2. THREE-TIERED ARCHITECTURE....28
- FIGURE 3. ARCHITECTURE OF THE TOLL ESTIMATOR....37
- FIGURE 4. PROCEDURE FOR USE OF MESSENGER....41
- FIGURE 5 - CORBA ARCHITECTURE....47
- FIGURE 6 - FIND CUSTOMER INITIALISATION OID....49
- FIGURE 7 - FIND CUSTOMER - CLIENT TO SERVER....50
- FIGURE 8 - FIND CUSTOMER - SERVER TO CLIENT....51
- FIGURE 9 - USE OF TABBED NOTEBOOKS....55
- FIGURE 10 - VISUAL PROGRAMMING....56
- FIGURE 11 - IMPLEMENTATION OF TABBED NOTEBOOK FUNCTIONALITY....57
- FIGURE 12 - FIND CUSTOMER USAGE....59
- FIGURE 13 - IDL COMPILING....60

1. Introduction

The topic of this thesis was first proposed by the management of IBM New Zealand. One of the major products of IBM in New Zealand is called the Integrated Customer Management System, or ICMS. This is a software package that is sold to telecommunications companies worldwide.

The ICMS product is currently undergoing a major upgrade which would see it converted from an RPG based application that is based on AS/400 type computers, to a C++ based application that would run on RS/6000 type computers (server) and Intel x86 computers (client).

What was proposed was a course of research that would study how Internet based technologies could be applied in the future use of this newly upgraded product.

2. Background

2.1 IBM New Zealand

IBM New Zealand is the sponsor of this thesis. Specifically, it is the Customer Care and Billing Development and Support Centre (CC&B) who have sponsored this research. The CC&B team in New Zealand comprises of about 600 people who are working on the development of current and future versions of ICMS.

The New Zealand development is being undertaken in three locations – Auckland central, Petone, and Christchurch. There is also development work being undertaken in IBM labs in France, Canada, and USA.

2.2 Integrated Customer Management System

As the name suggests, ICMS is an integrated computer system for managing customers. Customer Service Representatives (CSRs) of Telecommunications Companies normally use this application to process customer requests. For example, a CSR may use ICMS to order a new "call waiting" service, or to remove an erroneous phone call from a bill.

The ICMS application runs on AS/400 computers. These are large high performance computers that allow hundreds of simultaneous users to access it at any one time. The interface to the AS/400 computer is via a text console. Hence, the ICMS application is also accessed via the text console.

IBM is looking to upgrade the ICMS application to take advantage of modern technologies such as graphical interfaces (such as the Windows GUI) and also Internet Technologies. With these new technologies, the ICMS application will become more user friendly, and hence easier to use.

2.3 Internet technologies

The Internet is the name for a large collection of computers that have been networked together using a common standard for communications. It is used for messaging, information retrieval, entertainment, and many other services.

There are many technologies that power the Internet. All of these technologies have one thing in common – this is the use of the Internet Protocol. The Internet Protocol (IP) is a format for sending messages between computers. The other protocols that are in common use on the Internet build upon this protocol. For example, electronic mail uses IP to send the message between the source and the destination computer.

3. Non-functional requirements

Before dealing with the specifications or design of any software there are non-functional requirements that must be met. Non-functional requirements can be thought of as "how a task is achieved" rather than "what is the task". They can be thought of as the characteristics of the system rather than the purpose of the system.

In this chapter the desired characteristics, or non-functional requirements, of an Internet based Customer Support System are discussed and specified.

3.1 Performance

There are two types of measures that can be thought of in relation to the performance of a system - client performance and server performance.

3.1.1 Client Performance

There are three measures that have been identified as performance characteristics on the client. These are:

- Average time to display
- Average time till availability
- Average response time

Average time to display

The *average time to display* is the time it takes for the user interface to display some useful information (ie: content, not banners) after the user first presses the connect button in their browser.

A small amount of testing was done to get a rough idea of current display times. These results were obtained from a Pentium 166 running Netscape Navigator 4.0 connected to a LAN. The results are shown in Table 1.

• Table 1. Average time to display

	Site name	Time to display content (secs)	
		LAN	Modem (56 kbps)
	JavaWorld http://www.javaworld.com/	32	36
	Javasoft http://www.javasoft.com:8081/	12	22
	Eye-on-Objects http://eye-on-objects.com/	27	40
Standard	JDC http://developer.javasoft.com/	12	13
HTML sites	SE Using Java http://www.mcp.com/que/et/se_java2e/	17	23
	7am News http://www.7am.co.nz/	12	10
	Aardvark Internet news http://www.aardvark.co.nz/	9	13
	CNN Interactive http://cnn.com/	7	31
	ABC News http://www.abcnews.com/	43	45
	Map of Tallin, Estonia http://www.tallinn.ee/map/	42	21
Java applet	SlovakiaMap http://www.sanet.sk/	39	29
sites	Bonn City map http://cips02.physik.uni-bonn.de/~preusser/applets/stadtplan/plan_applet_d.html	30	36
	Virtual Boston http://www.pmg.lcs.mit.edu/~ng/Map/	38	44
	Netscope a http://www.merzcom.com/demos/netscape/demo.html	42	n/a

The times for HTML only sites tended to be around the 10 to 40 seconds region. The sites that took longer in loading tended to contain a larger number of graphics. Although graphics files add to the visual display of the pages, sites such as CNN and Aardvark showed that the use of different typefaces and colours can also be effective.

The times for loading of Java based sites were significantly longer. The *Map of Tallinn* site showed that simple interfaces can decrease the loading time. The *Netscope* demo site showed a small "please wait" and information window half way through loading.

The Customer Support System should aim to keep the content display time less than 30 seconds so that it is comparable to similar Internet sites.

Average time till availability

The *average time till availability* is the time it takes for the user interface to become available for use. This differs from the content display time since it is possible for some content to be displayed, yet the application is not fully available. For example, in the *Map of Tallinn,* the interface is loaded in 42 seconds, but the map does not display for another 11 seconds.

As in the section above, testing was done to get a feel for the average time to availability. The results are shown in Table 2.

• Table 2. Average time to availability

	Site name	Time to availability (secs)	
		LAN	Modem
	JavaWorld http://www.javaworld.com/	209	45
	Javasoft http://www.javasoft.com:8081/	51	33
	Eye-on-Objects http://eye-on-objects.com/	27	40
Standard	JDC http://developer.javasoft.com/	55	10

HTML sites	SE Using Java http://www.mcp.com/que/et/se_java2e/	20	29
	7am News http://www.7am.co.nz/	44	14
	Aardvark Internet news http://www.aardvark.co.nz/	44	24
	CNN Interactive http://cnn.com/	48	30
	ABC News http://www.abcnews.com/	76	48
	Map of Tallin, Estonia http://www.tallinn.ee/map/	113	21
Java applet	Slovakia Map http://www.sanet.sk/	355	29
Sites	Bonn City map http://cips02.physik.uni-bonn.de/~preusser/applets/stadtplan/plan_applet_d.html	65	36
	Virtual Boston http://www.pmg.lcs.mit.edu/~ng/Map/	62	56
	Netscope http://www.merzcom.com/demos/netscape/demo.html	238	n/a

Apart from the Javaworld site, the times for HTML based sites tend to range from 20 seconds up to a minute. The Javaworld site spent more of its time loading numerous advertising pictures.

The times for Java based sites take significantly longer to become available for use. As shown above, these sites ranged widely from 21 seconds up to almost six minutes. It is possible that poor network performance rather than an inefficient design may have effected the Slovakia Map site.

The Customer Support System should aim to be available within one to two minutes.

Average response time

The average response time is the time it takes for an application to respond to a request for information. As in the above sections, testing was done to give a rough comparison of similar applets on the Internet.

The results are shown in Table 3.

- Table 3. Average response times

	Site name	Average response time (secs)	
		LAN	Modem
	Map of Tallin, Estonia http://www.tallinn.ee/map/	15	9
Java applet	Slovakia Map http://www.sanet.sk/	60	40
Sites	Bonn City map http://cips02.physik.uni-bonn.de/~preusser/applets/stadtplan/plan_applet_d.html	40	18
	Virtual Boston http://www.pmg.lcs.mit.edu/~ng/Map/	35	9

As the above results show, the average response times are under a minute. There are some reasons why some of the above values are higher than others. The *Slovakia Map* applet had a long response time due to network lag to Slovakia. The Bonn City map had a higher response time because it was downloading a lot of data (ie: large bitmap graphics). The *Map of Tallin* applet was the fastest because it's map was a combination of bitmap and vector style, thus improving performance.

The Customer Support System should aim to have a quick response time by limiting the size of data that has to be transfered. A response time of less than 30 seconds would be adequate.

3.2 Ease of use

The kinds of people that the Customer Support System will be aimed are unlikely to have experience with the systems that a Customer Services Representative might have. Instead, they are likely to be everyday people who have a varying range of computer skills.

According to Sunsoft Distinguished Engineer Jakob Nielson, "the Internet doubles (in size) every year, and does done so ever since it was founded." Therefore, the Internet will grow by approximately 6% per month. This infers that about 1 in 18 visitors to a site have been using the Internet for less than one month.

A likely location for the Customer Support System would be a kiosk in a shopping centre. Potential users at this site would be even more unlikely to be experienced users since the normal requirement to own a computer is removed.

The implication is that many users are still learning how to use their web browser. Complex web based pages or applications are likely to overwhelm these less experienced users.

3.2.1 As simple as possible

"All Things Web" is a site devoted to Web designers and authors. The authors of the site state [1] that "simpler Web pages and simpler site designs are easier to master... especially for 'Net newbies'."

3.2.2 Other advantages of a simple design include

- More stable - less components for possible failure
- More compatible - less likely to use browser specific components
- Easier to maintain

3.3 Configurability

It is important that the Customer Support System be configurable. The intended customers will want to be able to change many of the settings for their own requirements. For example, a customer may want to apply their own corporate style to the applet – ie: the use of specialised fonts and colours.

4. Architectural Alternatives

In this section, a number of architectural alternatives for delivering dynamic information over a network will be covered.

The first option is to use a hypertext-based interface. This is the common interface seen on today's web browsers such as Netscape Navigator and Microsoft Internet Explorer. Aspects of both the clients interface and the web server and associated technologies will be discussed.

A second option explores possibility of using a more advanced interface based on the Java programming language. Java is a popular object-oriented language that can run in many differing hardware environments.

4.1 Hypertext based solutions

Hypertext Markup Language (HTML) is a language to specify the structure of documents for retrieval across the Internet using browser programs of the World Wide Web.

This section will focus on the architecture needed to deliver HTML based information to people using standard browser programs. A number of examples will be given of current systems that deliver dynamic information via these means.

4.1.1 Client Architecture

The client side of a hypertext-based system is simply a web browser program. This connects to a web server of some variety that downloads the hypertext files.

One of the problems that is likely to arise is selecting which features of HTML to use. Because of the competitive nature of the browser industry, new versions are released regularly with new and updated features. By using some or all of these new features, a web designer can effectively shut out a large proportion of Internet users who haven't upgraded to the latest new version.

There are two options to solving this problem. If a designer decides that the new features are important enough, they can develop and maintain multiple versions of their pages. Depending upon the amount of work required, this can be an expensive exercise. For example, at the time of writing, Clear Communications maintains three separate versions of their webpages - a Netscape version, an Internet Explorer version, and a plain Vanilla version.

The alternative is to develop an interface that will be compatible with the majority of web browsers. This seems to be the most used approach today.

Storing state information

A typical application of the Customer Support System might be requesting a new telephone for a household. The process might involve a customer entering in their customer number and possibly a password, selecting the phone they want, and finally entering in payment details. This process would probably involve at least three or four different screens.

The information that has been entered is known as state information. This needs to be held somewhere while the transaction is taking place. Some of it may also need to be sent to the server for each stage of the process. There are a number of ways in which this can be done.

Embedded in web page

The state information can be embedded in the web page by the serving process that created it. There are generally two different methods by which state information is transferred as a transaction progresses.

The state information may be embedded in a GET method. This is basically done by having a URL which has the state information stored in a similar manner to a DOS *command line parameter*. An example URL that entered a user's login and password might be:

http://www.news.co.nz/page_two?login=fred&password=barney

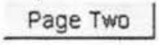
In the above example, a user is requesting page two of a news service, and passing login (fred) and password (barney) as state information.

State information can also be embedded in a form and sent via a POST method. The information is also entered by the serving process that created the HTML document. It is embedded into a form such as the one displayed below:

Source code:

```
<FORM METHOD=POST ACTION="http://www.news.co.nz/page_two">
<INPUT TYPE=HIDDEN NAME=login VALUE="fred">
<INPUT TYPE=HIDDEN NAME=password VALUE="barney">
<INPUT TYPE=SUBMIT NAME=submit VALUE="Page Two">
</FORM>
```

Output:



Page Two

The fields with the name *login* and *password* are both of the hidden type. As expected, they are not displayed. When the user presses the button that is labelled "Page Two", the data in these fields is sent to the web server.

The above two examples show the use of entering logins and passwords with little regard to security. If the above two examples were used in a real world example, it would be easy for someone else to discover the user id and passwords.

There are two main ways to protect the state data that is embedded in a web page. Firstly, this data should be scrambled or encrypted in some way. This will prevent someone from peering over the user's shoulder and easily recognising the login and password.

The state data must also be protected from a person who is browsing through the disk cache files (looking for logins and passwords). It is possible to force the web browser to not store a sensitive file in its cache by altering one of the fields in the HTTP response header. This header is generated for every request sent to a web server. The field that must be altered is the *expires* field, which states the date and time at which the sent file will no longer be current. If this field is set to the current date and time, then the file will not be stored in the cache. For example,

```
HTTP/1.0 200
Content-Length: 3495
Content-Type: text/html
Date: Tue, 15 Nov 1994 08:12:31 GMT
Expires: Tue, 15 Nov 1994 08:12:31 GMT
Server: CERN/3.0 libwww/2.17
```

Stored as a cookie

Persistent state information can be stored on the client using properties known as cookies.

"Cookies are a general mechanism which server side connections (such as CGI scripts) can use to both store and retrieve information on the client side of the connection. [3]" The cookie information is a string which is stored in a text file on the client.

There are two ways in which cookie information can be stored and retrieved. Cookie information can be sent using HTTP headers, or using JavaScript commands.

When a web client requests a file from a server, the serving process may set a cookie in its HTTP response by adding an extra field to the header. The *Set-Cookie* field is used to store information.

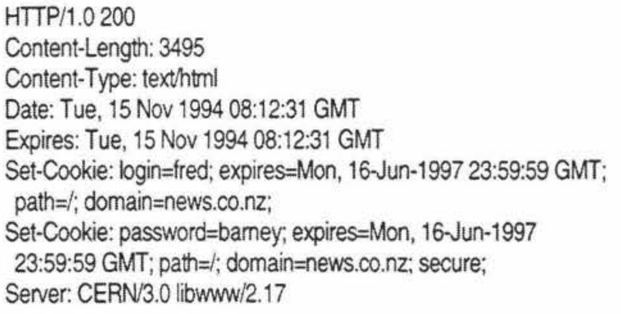
The information [4] that is stored consists of the following:

- A cookie label (name field)
- A cookie value (value field)
- An expiry date and time (expiry field)
- Name of the generating server's domain (domain field)

- Path attributes (path field)

The cookie label and value are self-explanatory. The expiry date and time indicate the time at which the cookie information will no longer be stored or given out. The domain field is used to identify generating server. The path attribute is used to determine when cookie information should be sent.

An example HTTP header is shown below:

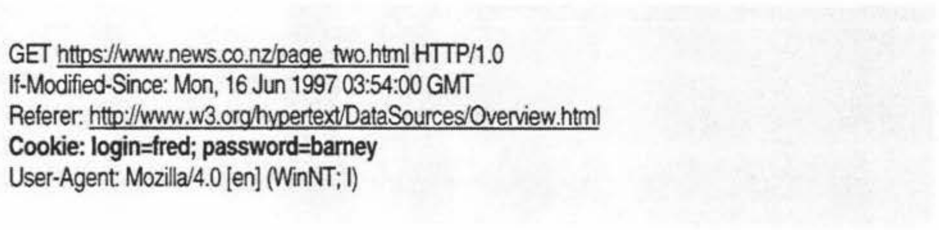


```
HTTP/1.0 200
Content-Length: 3495
Content-Type: text/html
Date: Tue, 15 Nov 1994 08:12:31 GMT
Expires: Tue, 15 Nov 1994 08:12:31 GMT
Set-Cookie: login=fred; expires=Mon, 16-Jun-1997 23:59:59 GMT;
 path=/; domain=news.co.nz;
Set-Cookie: password=barney; expires=Mon, 16-Jun-1997
 23:59:59 GMT; path=/; domain=news.co.nz; secure;
Server: CERN/3.0 libwww/2.17
```

The above example shows two cookies that set the login and the password of a user. The cookie will be deleted after the expiry date shown. Since the path attribute is set to the root directory, requests to any directory on the server will have this cookie attached. The domain news.co.nz indicates that any server in this domain will have this cookie information included in all requests. The password cookie has the secure attribute specified. This means that this cookie information will only be sent over a HTTPS (secure) channel.

Cookie information is automatically sent to a host whenever the domain entry and the path of a cookie match the location of a requested file. The cookie is sent as a field in the header of an HTTP request.

An HTTP request is shown below:



```
GET https://www.news.co.nz/page_two.html HTTP/1.0
If-Modified-Since: Mon, 16 Jun 1997 03:54:00 GMT
Referer: http://www.w3.org/hypertext/DataSources/Overview.html
Cookie: login=fred; password=barney
User-Agent: Mozilla/4.0 [en] (WinNT; I)
```

The above example shows the login and password being sent in an HTTP request header. Since the connection is secure (*https* in GET line), the password is sent.

JavaScript statements **[5]** can also be used to read and write cookie information. All of the fields that are used in HTTP headers are also used in the JavaScript statements.

Cookies are stored in JavaScript as a property of the document object. Setting cookies in JavaScript is similar to using the HTTP method. The *document.cookies* property is assigned a value of a new cookie. An example is shown below:

```
document.cookie = "login=fred";
document.cookie = "password=barney expires=Mon, 16-Jun-1997
 23:59:59 GMT; path=/; domain=news.co.nz; secure;";
```

The above example shows the login and password being set for a user. Since there are no properties specified for the login field, access would be set to the calling page only, and it would expire at the end of the session. The password property would be saved with all of the listed fields.

A JavaScript call to *document.write(document.cookie)* would produce the following result:

```
login=fred; password=barney
```

This shows the login and password of a user being accessed by a JavaScript call. Accessing specific cookies (rather than a list of them) is normally done with a user defined JavaScript function call. There are a number of books **[5]** that provide information on ways to do this.

As with state information embedded in web pages, cookies are also open to security hazards. The cookie information is stored in a file called cookies.txt (or MagicCookies on Macintosh based machines). This file is easily found, and able to be viewed by any user. Therefore, sensitive information such as passwords or personal data should be scrambled or encrypted.

Socket Encryption

In addition to the security measures that have been outlined for the storage of state information on the client, efforts should be taken to ensure that the data is not monitored while being transmitted to the host. Using an encrypted data stream can easily do this. A standard exists called Secure Sockets Layer (SSL) which encrypts the data sent between client and server. This will reduce the possibility of sensitive state information being acquired by an unknown third party.

4.1.2 Server Architecture

The architecture of systems that supply hypertext falls between two main types. One extreme is that the hypertext generator is implemented using a programming language of some type, and consists of a number of executable files. The other extreme is that the hypertext generator is created using a 3rd party visual tool, which generates the HTML according to the specified design.

Most common solutions available today fall somewhere between these two extremes.

Server connection

There are currently two popular ways in which a standard hypertext server can retrieve information from a 3rd party source. A connection can be made using either the common gateway interface (CGI) or via a server defined application programming interface (API).

Common Gateway Interface

"The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers. **[6]**"

There are two ways in which a CGI program is called. A user can make a POST type call or a GET type call. The factor that differentiates these two methods is the way in which the data is transferred from the server to the CGI application.

The POST type call extracts it's data from the standard input, while the GET type call extracts it's data from command line parameters. The web server also transmits information about the call through the use of environment variables.

Both calls then process the received data, and then generate HTTP and HTML code that is sent to the standard output. The hypertext server receives this data, and transfers it to the client.

A CGI type program can be written in almost any kind of language, since all it needs to be able to do is read / write from the standard input / output, and access environment variables and command line parameters.

Environments such as Pearl, and UNIX shells have been popular for CGI programming. For developers interested in higher performance, C/C++ is used for CGI programs. For even higher performance, developers must move from CGI type programming to API based applications.

Application Programming Interface

API type programs offer faster response and better performance [7] than their CGI counterparts. Hypertext servers that use CGI create a separate process for each request received. The more concurrent requests there are, the more concurrent processes created by the server. However, creating a process for every request is time-consuming and requires large amounts of server RAM. In addition, this can restrict the resources available for sharing from the server application itself, slowing down performance, and increasing wait times.

One way to avoid this is to convert the CGI program into a shared library. The server loads the library the first time a request is received. It then stays in memory, ready to service other requests until the server decides it is no longer needed.

Shared libraries are compiled into native machine code. The actual file produced depends upon the platform. Windows based machines use DLL files, while UNIX based operating systems use ".so" files.

These libraries must conform to the API that the hypertext server specifies. There are two common types of API available at the time of writing. These are the Netscape **[9]** Server Application Programming Interface (NSAPI) and the Internet Server Application Programming Interface (ISAPI) from Microsoft **[8]**.

Many of the server architectures outlined below offer the option of connecting to the hypertext server via the API interface.

Custom solutions

Custom solutions allow the developer the most flexibility of any of the options available. A developer may decide to develop their custom solution using a standard development environment, or they choose one of a number of tools designed specifically for this purpose.

A couple of example tools are outlined below. They attempt to reduce the workload by simplifying many of the tasks common to Internet based applications.

Perl

Perl is a common tool used for creating custom solutions. According to Perl's author Larry Wall, Perl "is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal)." "Perl is an interpreted language optimised for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information [10]."

Perl has been quite popular on UNIX based operating systems, and has been ported to many other platforms.

VisualAge WebRunner Server Works

WebRunner Server Works uses handlers and adaptors to transmit client requests and manage communications with a server. It can connect to the hypertext server via a number of methods, including NSAPI, ISAPI as well as CGI.

It reduces the burden of development by removing the developer's focus off the technology, and back onto the problem at hand.

3rd party solutions

3rd party solutions attempt to reduce the complexity of web development by introducing an extra layer(s) between the system and the developer.

The way in which this is achieved differs between applications. Examples of these layers range from macro based solutions up to purely visual programming models. Outlined below is a small sample of four different products that are available today.

Net.data

Net.data is a 3rd party solution that integrates a macro-based language into the HTML code. Although it targets databases, it is able to retrieve data from existing business logic as well (support for Java, Perl, REXX, C++).

Net.data makes its connection to the hypertext server through NSAPI, or ISAPI for high performance operation. It can also connect via CGI.

Livewire

Livewire is a tool provided by Netscape that uses server side JavaScript to develop dynamic web applications. As with Net.data, this 3rd party tool targets database access.

Livewire is specifically targeted at Netscape (Enterprise & Fastrack) servers, and is tightly integrated. Hence, Livewire based applications tend to be high in performance.

NetDynamics

NetDynamics is a visual tool for creating Internet based applications. This 3rd party tool focuses on database access. It is also able to extract data from external native applications.

NetDynamics also allows developers to write their own Java class files through the use of a NetDynamics API.

NetDynamics makes it's connection to the hypertext server through NSAPI, or ISAPI for high performance operation. It can also connect via CGI.

Lotus Domino.Connect

Domino.Connect is an add-on tool for Lotus Domino (hypertext server) that allows access to databases and legacy systems. Domino.Connect is used in conjunction with a scripting language called LotusScript.

Because Domino.Connect is an add-on for the Domino hypertext server, there is no requirement to connect via CGI. This makes this 3rd party tool useful for high performance applications.

4.1.3 Examples

The examples in this section outlines practical situations of organisations that are using a hypertext based interface to supply dynamic applications on the Internet (or their own Intranet).

Central Power, Palmerston North

Central Power is a Palmerston North based power company. They allow their customers to query their own accounts. This ability is implemented using Perl and this connects to an Informix database.

State information is embedded in the HTML code using both GET and POST. This information includes both customer ID number and PIN number. It is not encrypted or scrambled in any way.

The pages that are sent to the client are not encrypted either. That is, the server does not use SSL to create a secure channel between client and server. This makes it possible for a 3rd party with the ability to monitor network traffic to pick the sensitive details out of the byte stream.

The HTML pages that are sent to the client are not set to expire, and are therefore stored in the local cache. It is therefore possible for a person to search the local cache for the phrase "My Account," identify the file, and then read the customer number and PIN number.

4.2 Java based frontend

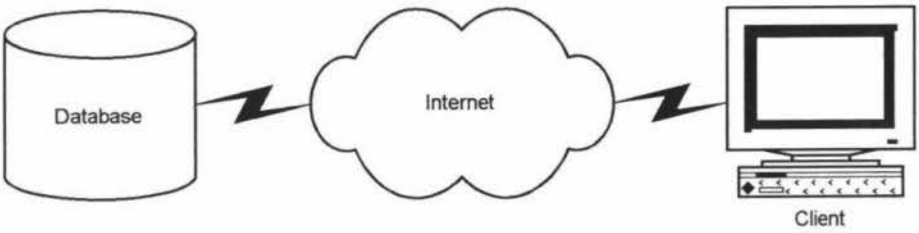
Java is a modern programming language that was developed by Sun Microsystems and first released in December of 1994 [1]. Code that was been written in the Java code language is compiled and then run in what is known as 'The Java Virtual Machine (JVM).' The JVM has been ported to many different platforms including such as Windows 32, Windows 3.1, OS/2, Macintosh, various UNIX OS's, and many others.

A version of the JVM has been implemented in some of the more popular web browsers such as Netscape Navigator, Microsoft Internet Explorer, and others.

This section will focus on the client implementation as well as the architecture needed to support a Java based frontend.

4.2.1 Two-tier model

The two-tier architecture contains two computers - a client and a server. The - system architecture is shown in Figure 1 below:



• Figure 1. Two-tiered architecture

The three components of the application - the model, the view, and the controller are divided among the two tiers. The view is implemented on the client. The model is stored on the server, and the controller is split between the two tiers. The bulk of the controller logic is stored on the client. A small portion of the controller logic is implemented on the server. This includes functionality such as integrity checks, querying, and database management. This is shown below in Table 4.

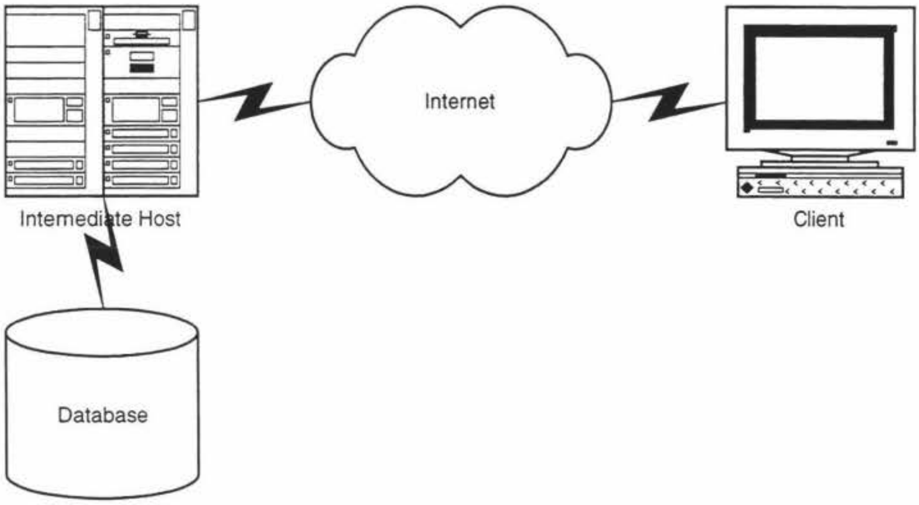
	Server	Client
Model	ALL	NONE
View	NONE	ALL
Controller	Integrity checks Querying Database management	Application logic

• Table 4. Distribution of application on 2-tiered architecture

Requests to the server from the client would typically be done through the use of the [11] Standard Query Language (SQL). SQL based queries are normally packaged in a JDBC (Java Database Connectivity) call when using the Java language.

4.2.2 Three-tier model

The three-tier model contains three computers - a client, a database server, and an intermediary server. The system architecture is shown in Figure 2 below:



• Figure 2. Three-tiered architecture

The three components of the application - the model, the view, and the controller are divided among the three tiers. The view is implemented solely on the client, while parts of the model and controller are distributed over all three tiers.

A small portion of the controller logic is implemented on both the client and the database. The majority of the logic is implemented on the intermediary host. A compromise has to be made to decide how much logic will be distributed between the client and intermediary host. The balancing factors include:

- Minimising the number and size of class files required to be downloaded to the "thin client."

- Minimising the amount of network traffic that must pass between client and intermediary host.

So to solve the above issues, simple procedures such as manipulating displayed data, or parsing input should be stored on the client. CPU intensive or larger procedures should be stored on the intermediary server. An example of this would be a procedure to retrieve data from the database.

Advantages

Calls made from the client to the intermediary server provide greater overall flexibility than the SQL type queries made in a two-tier type architecture. This is because the client simply passes the parameters needed for the request and accepts the returned values specified in the call.

Unlike the two-tier implementation, the three-tiered client does not need to understand SQL. This means that changes can be made to the underlying database, or it's associated technologies, without any changes being made on the client.

Having separate software entities allows the individual tiers of the system to be developed in parallel.

Middle-tier class libraries may be re-used by other applications.

Disadvantages

The three-tier model brings an increased need for network traffic management, server load balancing, and fault tolerance.

4.2.3 Communication

There are a number of ways in which external clients may communicated with the middle tier. These range from a simple sockets based implementation to a standards based systems like CORBA and RMI.

Remote Method Invocation

Remote Method Invocation (RMI) is a Java based standard that allows Java based clients to make calls to remotely located Java based server applications. RMI uses Internet Protocol as its transport mechanism.

Creating an application that utilises RMI is a relatively simple process for a competent Java developer. Outlined below are the steps needed to create an RMI based application.

1) Define a Java interface that models the class on the server.

2) Build a class that provides an implementation of all of the methods listed in the above definition. Define a simple constructor that this class in the RMI Registry.

3) Use a Java tool called *rmic* to automatically generate "stubs" and "skeletons" for use on the client.

4) Create a client program that uses the new objects.

5) Start the Java RMI Registry, run the server, and start the client.

The RMI Registry is a program that manages connections to the server classes.

The advantage of using RMI is that it is built into the Java environment, and its use is fairly straightforward. The disadvantage is that RMI is a Java only technology, and hence it cannot be used to communicate with non-Java based systems.

The Java RMI based communication middleware will be investigated and outlined in a case study.

CORBA

Common Object Request Broker Architecture (CORBA) is a messaging system that allows different hardware and software packages to interoperate. "Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them" [13].

CORBA uses a language called IDL (Interface Definition Language) which is used to define the interface of the application, or object that is available for use. For example, if a C++ program had a class called "Customer" (see below).

```
class Customer {
            String firstName;
            String lastName;
            int Age;
}
```

The above customer class has three attributes – the first and last name of the customer, and his or her age. An IDL interface could be written to describe these attributes.

```
Interface Customer {
            attribute string firstName;
            attribute string lastName;
            attribute int Age;
}
```

As can be seen, the above IDL interface describes the C++ object called Customer. This interface can now be "compiled" into any CORBA compliant language that needs to access the C++ class.

For example, the above interface will be compiled into a Java "skeleton" as shown below:

```
public interface Customer {
  public String get_firstName();
  public void set_firstName(String value);
  public String get_lastName();
  public void set_lastName(String value);
  public String get_address();
  public void set_address(String value);
}
```

This interface can now be called from a Java client.

To bind the different environments together, CORBA makes use of a middleware program called an Object Request Broker (ORB). The ORB takes a request from one environment, translates it into a common format, and passes it to another ORB in the different environment. This ORB then makes the request, and passes the results back in the same manner. In the example above, the ORB will allow the Java client to make calls to the C++ server to access the data on a customer.

The CORBA communications middleware will be investigated and outlined in a case study.

5. Toll Estimator case study

A Toll Estimator is a program that can be used to estimate the approximate cost of an intended long-distance toll call. This would be a useful application for customers of a telephone company, as they would be able to find out the price in advance of making a call.

This application was chosen as a suitable case to study, as it would involve the use of a combination of many Internet based technologies. For example, the program is expected to be run on a customer's computer, but it will need to communicate with a database that is located at the telephone company.

5.1 Background

The Toll Estimator was developed to investigate the use of Java based technologies which could be used through a standard web browser. It was hoped that the skills and knowledge that were gained through the development of such a program could be used to develop future components of a possible Internet solution for ICMS.

The functionality of toll estimation was chosen for this program because of the skill set that was already in existence at IBM. Two separate Toll Estimation programs had been written in the past. The first Toll Estimator was written in the Smalltalk language, and this was then rewritten in the C++ language.

It was hoped that these skills could be called upon when needed during the development of the Java version of the Toll Estimator.

5.2 Intended use

The Toll Estimator program was aimed at an everyday telephone user. It was hoped that they might be able to connect to the Internet, and then start up their web browser application. They would enter in the web address of their local telephone company, and select an option that would then start up the Toll Estimator program.

It is quite unlikely that a person would go to all this trouble to simply check on the price of a call. The reason for the development of this program was not to produce a useful tool, but to investigate the use of these technologies so that they could be applied to develop a tool that consumers might find useful.

5.3 System Architecture

5.3.1 Options

There were basically two broad options that were possible for the architecture of the Toll Estimator. These were either a two-tier or three-tier architecture.

A two-tier architecture would involve a "fat client" connecting directly to a database. A three-tier architecture would involve a "thin client" that would connect to a middle tier, which would then connect to the database.

The 3-tier architecture was chosen for the reasons outlined below.

5.3.2 Minimising the startup time

The Toll Estimator is aimed at domestic customers. These people tend to connect to the Internet via slow speed connections. For example, the maximum possible connection speed is 56 kbps, but the majority of people would use slower connections such as 14.4 or 33.6 kbps.

The amount of data to be downloaded to make the applet run will increase the startup time of the applet. The number and size of class files must be kept to a minimum.

In a two-tier architecture, various components such as database drivers would have to be downloaded. A potential user of the Toll Estimator may not have the patience to wait for the entire program to download. The three-tier architecture would leave the database drivers on the middle-tier. This would reduce the amount of data that must be downloaded, and hence minimise the startup time of the applet.

Packaging and compressing the files

Another way to reduce the startup time is to compress and package the files into a single file. Personal experience has shown that the number of files to download can often make more of a difference to the startup time than the file's size.

The files that are downloaded are often small in size. For example, a typical Java class file would probably be no greater than 5 kB. Larger class files might reach up to 20 kB, but this would be unlikely.

If the user were using an average modem, then the data could probably be transferred in a couple of seconds. Because of the nature of the Internet, delays normally occur when requesting files. This generally adds a couple of seconds before the data starts to be received. Hence, this small 5 kB file might take around four or five seconds to download.

Java class files tend to be downloaded in a sequential manner. This also adds to the delay of downloading Java applets. If an applet has around 10 class files, then a user could expect to wait for almost a minute for an applet to start up. On a day of unusual congestion the delay would be even longer. Obviously this is not acceptable for a simple Toll Estimator, although users may be willing to wait longer for applets which provide a more useful service.

Packaging the Java classes into a single file can vastly reduce this delay. Popular web browsers from Netscape and Microsoft both support this file packaging method. The Netscape browser uses a compressed format known as a JAR file. The Microsoft browsers also used a compressed file format – the cabinet (or CAB) file.

The result would be a single file that now contains all of the Java class files. With the addition of the file compression, the total volume of data to be downloaded is now reduced. In the example above, the ten 5 kB class files might be compressed into a 30 kB file. With a delay of around two seconds and a download time of around 10 seconds, the total waiting time would be reduced to less than a quarter of a minute. This is likely to be within the patience threshold of the average user.

5.3.3 Java security restrictions

The Java language has a number of features built into it that protect the environment of the end user. These security features apply only to those Java programs (applets) that run inside a web browser. This is necessary because Java applets are often run without the prior knowledge of end users. If an applet had unrestricted access to the environment in which it was run, then it could possibly access or destroy sensitive data.

There are two main restrictions that apply to Java applets. The first restriction involves access to data within the environment in which it is run. For example, an applet may not read or write files stored on a user's disk drives. This would prevent a hostile applet from wiping a user's hard disk.

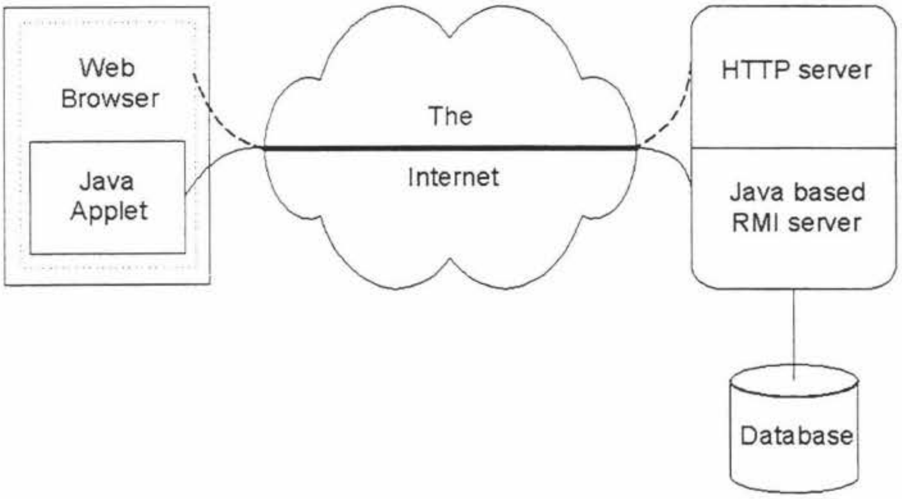
The second restriction involves access to the network. Java applets may not make connections to any other computers. An exception to this rule allows the applet to connect to the host from where it was downloaded. This restriction stops the applet from becoming a nuisance to other network users.

This second restriction could create a problem for the Toll Estimator if a two-tier architecture was chosen. This is because the Toll Estimator needs to connect to a third computer that contains the pricing information. A three-tier architecture would be able to work within the rules Java security model, since the middle tier would be located on the same computer as the web server. Since the middle tier would be a Java application (ie: not an applet), it would have unrestricted network access, and hence would be able to connect to the database for pricing information.

5.3.4 Architecture model

The 3-tier architecture model is shown below in Figure 3. The Java applet runs inside the web browser. The applet makes calls to a RMI based server located on the middle tier. This makes Java methods available for client applets to call. These methods make SQL type queries to the database.

The Java applet runs inside the web browser. The applet makes calls to a for client applets to call. These methods make SQL type queries to the database.



• Figure 3. Architecture of the Toll Estimator

5.4 Client Architecture

There are three major components to the client side architecture. Each component runs in its own thread. These are the startup component, the user interface component and the messaging component.

5.4.1 Startup component

The startup component is the simplest of the three client-side components. The tasks that it performs are:

- Instantiate (create and initialise) the user interface and messaging components.
- Extract applet parameters

- Start and stop component.

When the applet is first started, the user interface and messaging components are instantiated.

The startup component will extract the parameters that the user defines in the HTML page. The parameter extracted is the URL that points to the RMI server. This URL is passed to the messaging component on instantiation.

The other task of the startup component is to start and stop the other two components. The start and stop tasks are performed when the end user performs actions such as minimising the window, or moving off the web page in which the applet is stored. By stoping the components when the applet is not in use, the end user's machine will not be task-loaded for no purpose.

5.4.2 User Interface component

The user interface (UI) component provides all the visual needs of the Toll Estimator.

The UI was implemented using a class called TollFrame (see section 8.1.8 for documentation on this class). This class provides all the visual needs of the Toll Estimator applet.

When the class is first instantiated, the on screen components are displayed. Initially, these components do not display any data in them. When the class's start method is called, these on-screen components are then filled.

After the user interface has been drawn, this component lies dormant. It waits for events generated by the user. The three events that are monitored are "pressing Calculate," "pressing Reset," and "pressing Close."

If the calculate button is pressed, then the values of the on-screen components are parsed - checking for validity. If there are any problems, then an error is displayed, otherwise the request is sent to the messenger component.

If the reset button is pressed, then all of the values displayed on the screen are reset to their default values.

If the close button is pressed, then the window is hidden (it gets automatically destroyed when the web browser moves to another page).

The UI also waits for responses from the messenger. The two responses that occur are setCallCost and setLocations. These two messenger generated events will be discussed in section 5.4.3.

5.4.3 Messaging component

The messaging component provides other objects with the facility to request data from a remote server. The messaging component is implemented using a Messenger class. This class is implemented in a separate thread so that it does not halt other tasks while it makes remote calls.

The Messenger uses two public methods to supply information to external classes. The information that the Messenger can supply is a list of calling locations, and also call costing.

To achieve a threaded nature, the Messenger has a loop that is regularly checking two vectors for new entries. One vector named callCostRequesters is filled by calls to the requestCallCost method. The other vector named locationsRequesters is filled by calls to the requestLocations method.

When the loop detects that a request has been made (1 or more items found in the vector), it makes a call to executeCallCostRequest or executeLocationsRequest (according to which vector contains the items). These methods extract the information from the vector, process the request, and send the results back to the external class.

Processing the request is a simple task for the Messenger. When the Messenger is first started, a Remote Method Invocation (RMI) connection is made to the middle tier. This connection is then used by the executeCallCostRequest and executeLocationsRequest methods whenever a request is made.

5.4.4 Communication between components

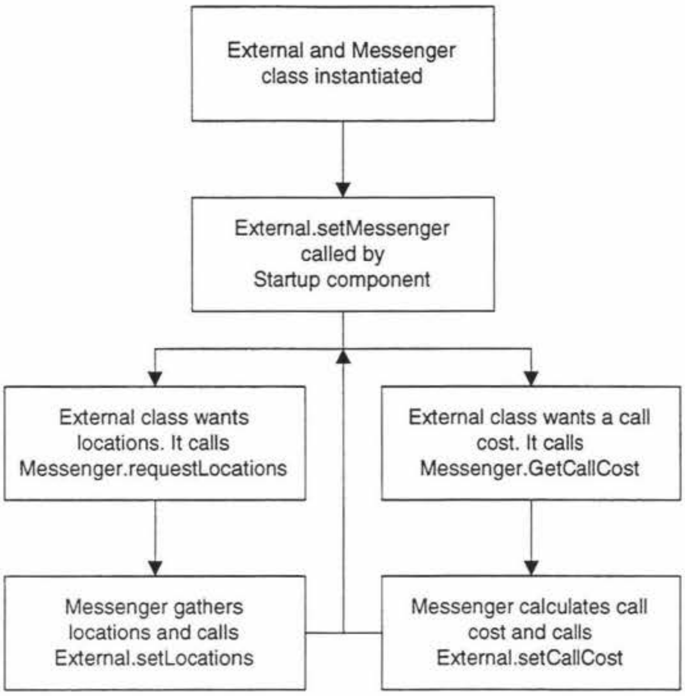
External classes that want to use the facilities of the Messenger will do so through call-back style requests. To allow this to take place, a Java interface was written. This interface (see section 8.1.1) defines three methods - setCallCost, setLocations, and setMessenger. These must be implemented by classes that use this interface.

The setMessenger method is called by the startup method. This method gives the external class a reference to the Messenger object that it will use.

The Messenger object calls the setLocations and setCallCost methods. These two methods get called when the Messenger has information to deliver to the external class.

The Messenger class has two public methods that are used to request information. These are requestCallCost and requestLocations. An external class will call these to make requests for information.

The procedure for making calls to the Messenger is outlined in Figure 4.


External and Messenger class instantiated
External.setMessenger called by Startup component
External class wants locations. It calls Messenger.requestLocations
External class wants a call cost. It calls Messenger.GetCallCost
Messenger gathers locations and calls External.setLocations
Messenger calculates call cost and calls External.setCallCost

- Figure 4. Procedure for use of Messenger

5.5 Message passing

The callback design makes the notification of errors more difficult, since any errors that may occur will not happen during the actual call. This means that Exceptions cannot be used to notify the external class of error conditions. To solve this problem, a special class was developed to pass both data objects and error messages.

The Message class is used for passing results between the middle-tier, the Messenger, and the User Interface.

What makes this class special is that it also provides a way in which error messages may be passed without throwing exceptions. This class was created to solve the problem of indicating the presence of an error to a callback function. Each message may contain one object and any number of error messages.

The class contains methods for testing for the presence of any errors that may have occurred and extracting both the objects themselves and the errors.

5.6 Middle tier Architecture

The purpose of the middle tier is to provide objects that can be remotely accessed by applets. This is achieved by binding an object called Databaselmpl to an RMI registry.

One the object is bound, it can be easily access from remote clients such as applets.

The two methods in the Databaselmpl object that are used are getLocations, and getCallCost.

The getLocations method makes a connection to the database server and extracts all of the call locations. These then get placed into a Vector and sent back to the client. If there are any errors, then these get sent along too.

The getCallCost method takes call information parameters in its header, and uses these to make a query to the database. The result is then used to calculate the cost of the call. The costing information is then sent back to the client, along with any errors that may have occurred.

5.7 Server Architecture

The database server that is used is DB2 for Windows NT. In a production version of the Toll Estimator, DB2 would be run on either AIX or AS/400.

The tables that were used in this prototype include:

CITIES - this table contains the location names and numbers. For example:

```
"Auckland",1

"Middlemore",2

"Papakura",3

"Pukekohe",4
```

CITYSTEPS - this table lists the charging step for calls between two locations. For example:

```
1,2,"A"
```

```
1,3,"B"
1,4,"C"

1,5,"D"
```

STEPPRICES - this table lists the cost per minute of the charging steps. For example:

```
"A",.05

"B",0.09

"C",0.13

"D",0.15
```

The data in these tables is accessed using queries to a JDBC (Java database connectivity) class. This makes a direct connection to the database to execute the queries.

5.8 Future extensions

5.8.1 Data source

As described above, the data is obtained from a local DB2 database. An extension to this project would be to add the capability to connect to a live database. For example, the data could be obtained from the ICMS database on the AS/400. This would add realism to the project.

5.8.2 Create Java Beans

A Java Bean is a set of one or more Java classes that is defined in a standard manner so that other applications may easily use it. For example, a textfield bean may have a property called contents. Being a Java bean, it would then have two methods called setContents and getContents.

As well as properties, beans also have events and methods. An example of an event that might occur could be "button pressed" or "e-mail received". An example of a method might be "sendMessage" or "convertToDollars".

The Toll Estimator could be converted into a Java Bean. This would allow it to be used to build a part of a more complex Customer Services Java program.

6. Find Customer case study

The aim of this project was to build a Java based client application that would demonstrate practical communication with an OS/2 based C++ server. The application should demonstrate the use of modern Java technologies such as Java Beans and the new JDK 1.1 event model.

6.1 Options for the communications architecture

6.1.1 Requirements

The broad requirement for the communications architecture was to enable an application written in version 1.1 of the Java programming language, to make calls to C++ based objects residing on a remotely located OS/2 workstation.

Other requirements included:

- Adequate performance – the overall application should perform at about the same speed as could be expected from a similar application. For example, an Internet based search engine would be a similar application.

- Scalability – the application should be based upon technologies that are able to scale to many numbers of users, without significant performance drop-offs.

- Reliability – the application should be able to operate continuously for a long period without crashing.

- Ease of use – a potential user should be able to understand the application within a few minutes of using it. The application should be based upon industry standards for user interface design.

- Development efficiency – the development of the application should be undertaken so that reuse of code is maximised.

6.1.2 Three options

Based on the above requirements, there were three main options that were considered. These were:

- A "do it yourself" type sockets design

- Hybrid Java RMI and Java JNI

- CORBA

6.1.3 Sockets

The socket design was never really considered as a practical option. This was because it would basically be "reinventing the wheel." Also, considering this project was to be a "proof of concept", we didn't think we would be proving anything that hadn't already been proved by anyone else.

It was also thought that it was important to keep to industry standards, and developing our own proprietary design would not conform to this aim.

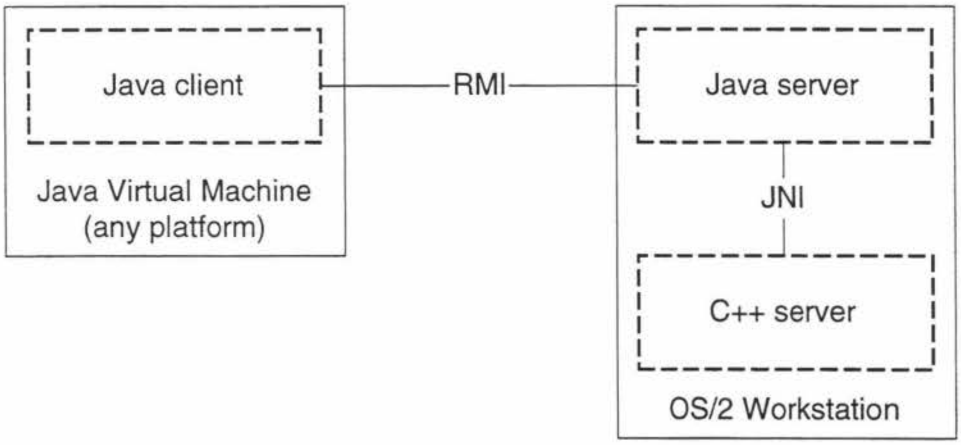
6.1.4 Java RMI and Java JNI based middleware

As described earlier, Remote Method Invocation (RMI) is a communications mechanism native to Java. It enables communication between remotely located Java clients and Java servers. Unfortunately, it cannot communicate with programs written in other languages.

Java Native Interface (JNI) enables Java based applications to make calls to software that has been written in other programming languages. The client programs must reside on the same platform as the server.

The aim of the hybrid design was that the Java client would use RMI to communicate with a Java based server located on the OS/2 workstation. This would pass the requests onto the C++ server (also on the OS/2 workstation) via the Java Native Interface.

The architecture diagram is shown below:



This design was not chosen, and there are a number of reasons why.

Inadequacies of JNI

One of the major reasons is due to the inadequacies of JNI. Features such as the lack of support for multiple inheritance, exceptions, and templates would have made creating a work-around relatively difficult and time consuming.

Performance of the Java server

Another factor in the decision was the question of performance. Java based applications are known to have difficulty matching the performance of C++ applications. This is due to the interpreted nature of the language (compared to C++ that is compiled).

The number of simultaneous clients that could be supported by one server would be significantly less than a pure C++ only based server.

Risk

The risk of failure would be higher since there would be one more new technology to become aquatinted with. These were being both RMI and JNI, when compared with only CORBA.

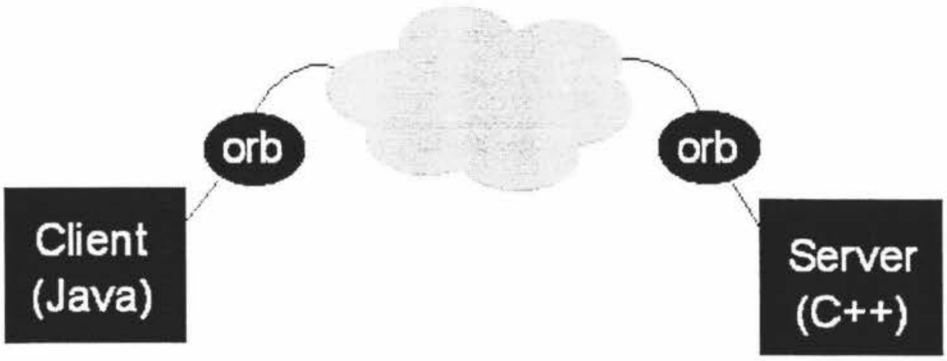
We were also uncertain of the future of RMI. At the time, there were rumours in the industry that Microsoft would not be supporting RMI in its upcoming version of Internet Explorer. Although the project had little to do with web browser technologies at the time, one of the future extensions of the project would be to create a web browser based version. As it turned out, the rumour was true.

Another rumour was that Sun Microsystems was speculating about the possibility of converting RMI so that it would use the same communication protocol (IIOP) as CORBA. This seemed like another reason to choose CORBA as the communications middleware.

6.1.5 CORBA based middleware

As described earlier, CORBA is a communications middleware standard defined by the Object Management Group. The overall architecture model is similar in function to the hybrid design described above, but has removed one of the layers on the server.

The architecture diagram is shown below:



- Figure 5 - CORBA architecture

As you can see in Figure 5, the two Object Request Brokers (ORBs) connect the Java client and the C++ server directly to each other.

Performance

The CORBA based architecture is expected to have an improved performance because of the removal of the server-side Java layer. The ORB that has taken its place is a C++ library and is tightly integrated with the C++ server.

Likewise, the ORB on the Java client consists of Java classes and is tightly integrated with the client.

Risk

The CORBA implementation has replaced the combination of both RMI and JNI. This means that there will be one less technology to learn.

Functionality

CORBA is a large architecture that consists of many different services. These range from simple naming services to more complex encryption and authentication services. Although the use of such services as encryption and authentication were not in the objectives of the use case, their use was thought to be of importance in the future.

Many of the limitations of the hybrid RMI and JNI were resolved in the CORBA architecture. For example, exceptions generated by the C++ server would flow across to the Java client.

6.2 Building the communications architecture

A communications middleware based upon the CORBA 2.0 standard was chosen.

6.2.1 Supplier

Iona Technologies of Dublin, Ireland was chosen as the supplier of the CORBA implementation. Iona's Orbix is a CORBA compliant ORB with bindings for many different languages and platforms. At the time of writing, Orbix was the only supplier that had an operating ORB with C++ bindings for OS/2.

The two products chosen were OrbixWeb (Java ORB) and Orbix for OS/2.

6.2.2 Defining the interface

The CORBA 2.0 standard defines IDL (Interface Definition Language), which is used to define the interface between the two ORBs.

Creating an IDL file

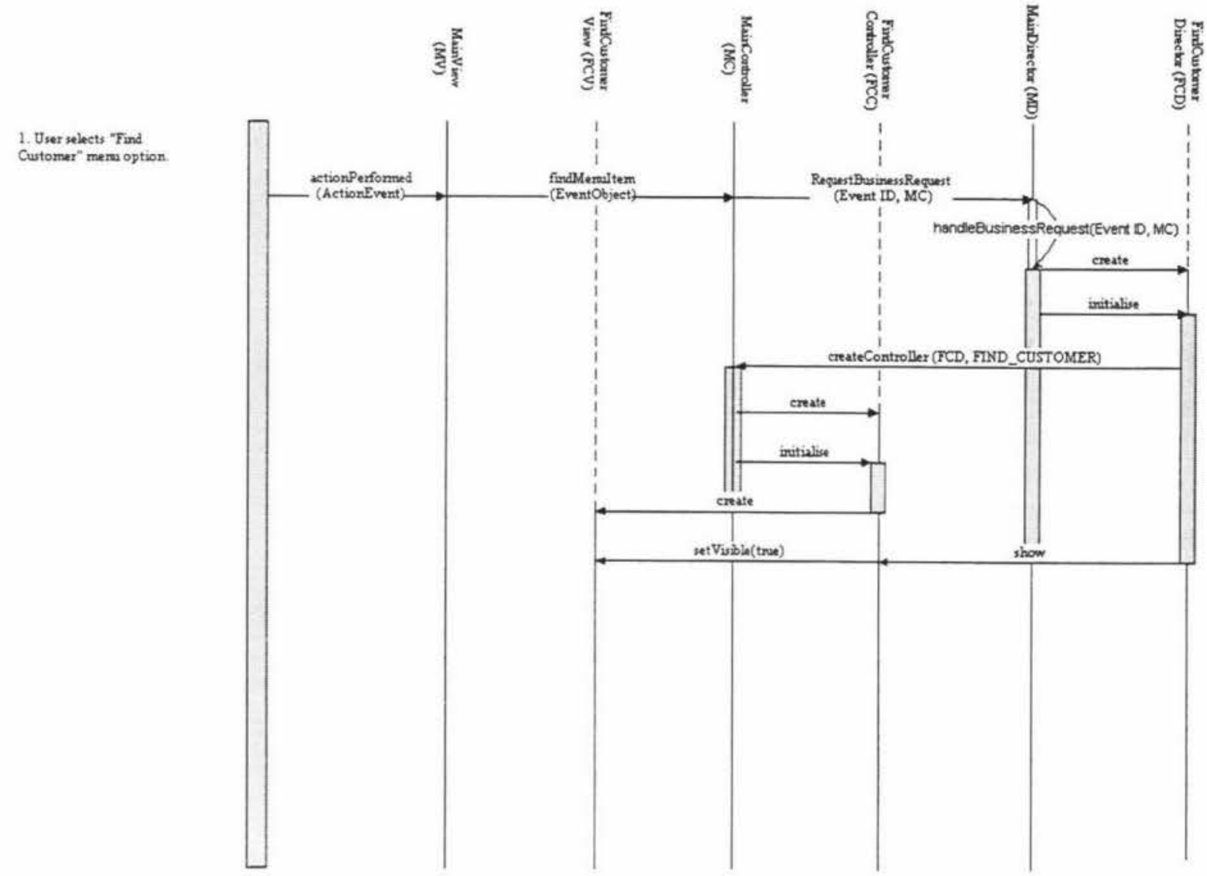
To create the interface definitions files, we first had to analyse what would be included in the content. This was done mainly with the help of Object Models and Object Interaction Diagrams (OID).

An Object Interaction Diagram is a drawing that outlines how messages and events are passed between different objects in a computer system.

Because our task was to replicate the functionality of the C++ based Find Customer use case, we started with the C++ OIDs. There were three OIDs for the C++ based use case.

These three OIDs were then converted to a Java style - for example, since the client was to be using Java Beans, many of the C++ events styles were updated for the Java Event model. The first OID (see Figure 6) outlines the initialisation of the client. The initiating event was the selection of a menu item called "Find Customer". This then creates and initialises a new FindCustomerDirector object, which sets up the new FindCustomerController and FindCustomerView objects. The view is then displayed.

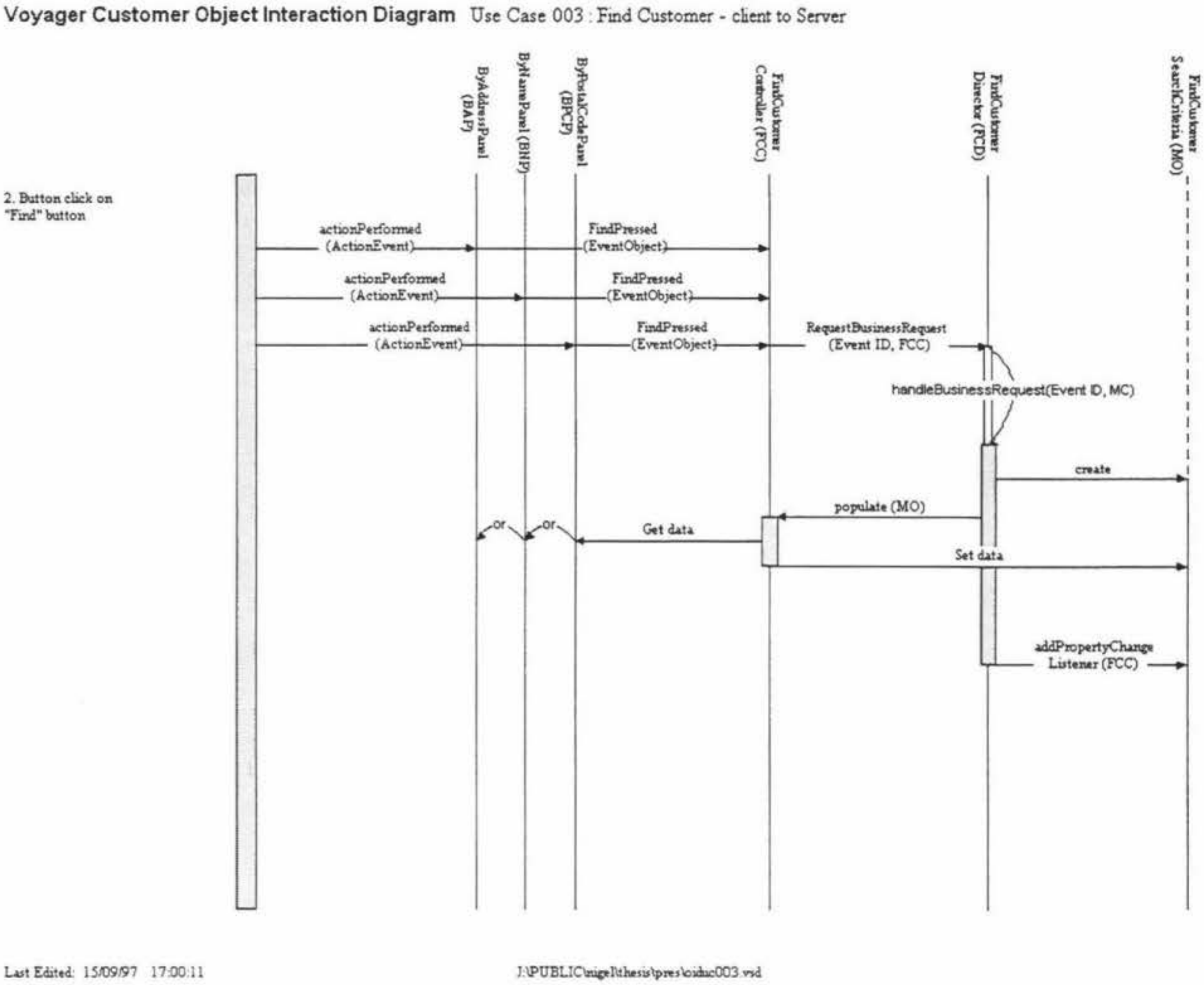
Voyager Customer Object Interaction Diagram Use Case 003 : Find Customer


MainView (MV)
FindCustomer View (FCV)
MainController (MC)
FindCustomer Controller (FCC)
MainDirector (MD)
FindCustomer Director (FCD)
1. User selects "Find Customer" menu option.
actionPerformed (ActionEvent)
findMenuItem (EventObject)
RequestBusinessRequest (Event ID, MC)
handleBusinessRequest(Event ID, MC)
create
initialise
createController (FCD, FIND_CUSTOMER)
create
initialise
create
setVisible(true)
show

Last Edited: 15/09/97 17:00:11

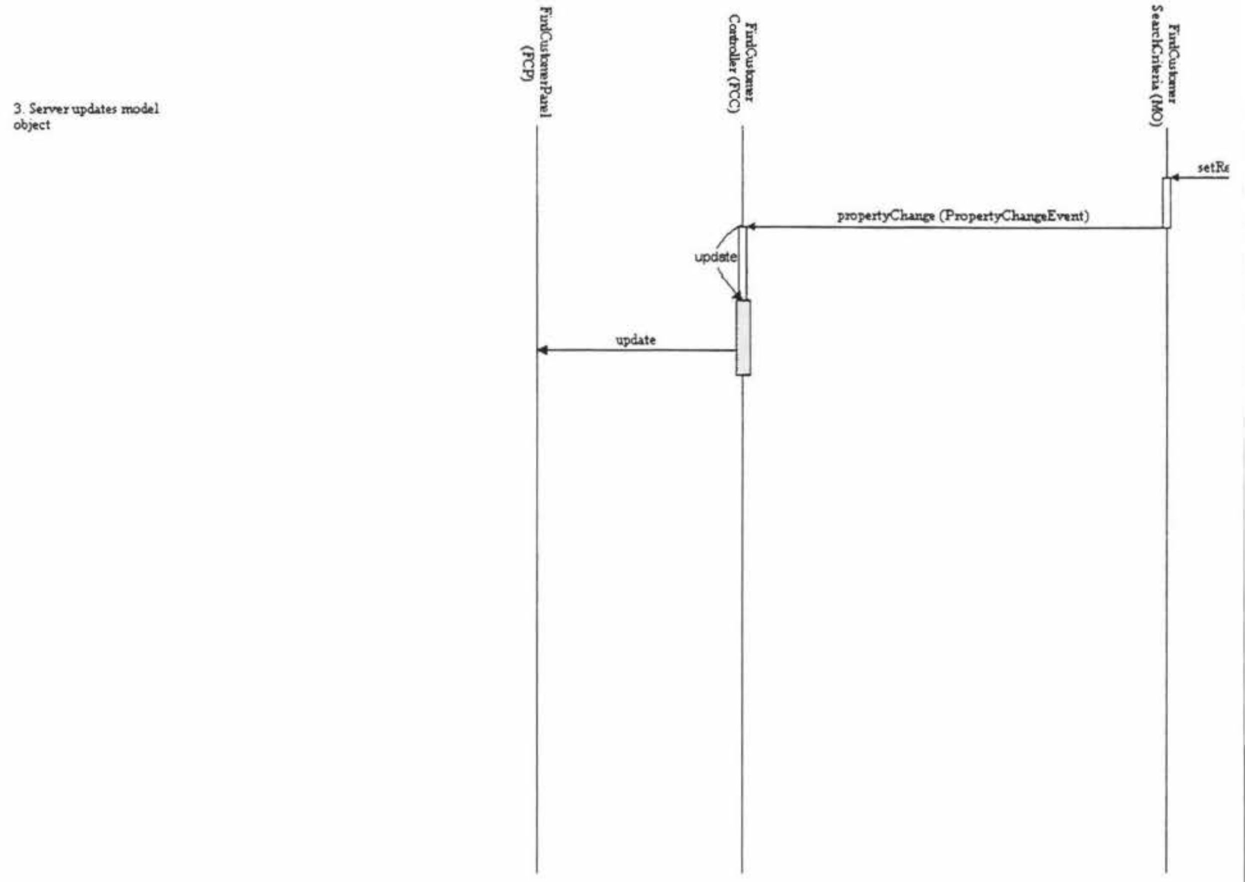
J:\PUBLIC\nigel\thesis\pres\oiduc003.vsd

- Figure 6 - Find Customer Initialisation OID


Voyager Customer Object Interaction Diagram Use Case 003 : Find Customer - client to Server
ByAddressPanel (BAP)
ByNamePanel (BNP)
ByPostalCodePanel (BPCP)
FindCustomer Controller (FCC)
FindCustomer Director (FCD)
FindCustomer SearchCriteria (MO)
2. Button click on "Find" button
actionPerformed (ActionEvent)
FindPressed (EventObject)
actionPerformed (ActionEvent)
FindPressed (EventObject)
actionPerformed (ActionEvent)
FindPressed (EventObject)
RequestBusinessRequest (Event ID, FCC)
handleBusinessRequest(Event ID, MC)
create
populate (MO)
or
or
Get data
Set data
addPropertyChange Listener (FCC)
Last Edited: 15/09/97 17:00:11
J:\PUBLIC\nigel\thesis\pres\oiduc003.vsd

- Figure 7 - Find Customer - client to server

Voyager Customer Object Interaction Diagram Use Case 003 : Find Customer - server to client


3. Server updates model object
FindCustomerPanel (FCP)
FindCustomer Controller (FCC)
FindCustomer SearchCriteria (MO)
setRe
propertyChange (PropertyChangeEvent)
update
update

Last Edited: 15/09/97 17:00:11 J:\PUBLIC\nigel\thesis\pres\oiduc003.vsd

• Figure 8 - Find Customer - server to client

The spilt between client and server is between the controller objects and the director objects. In the above diagram, there are three events that traverse the split. These are

1. A call to the Request Business Request method of the Main Director
2. A call to the create Controller method on the Main Controller
3. A call to the show method of the Find Customer Controller.

The first operational attempt at a CORBA interface is shown below in Table 5:

```
// *********************************************************
// * IDL for Network Computing proof of concept *************
// *********************************************************
//
// file: POC4.IDL
// First created: 29 September 1997

requestBusinessRequest

    interface Customer {
       attribute string firstName;
       attribute string lastName;
       attribute string details;
    };

    typedef sequence<Customer> CustomerList;

    interface FindCustomerSearchCriteria {
       attribute string searchName;
       attribute CustomerList searchResults;
    };

    interface FindCustomerViewControllerInterface {
       oneway void populate( in FindCustomerSearchCriteria fcsc
);
       void update( out  FindCustomerSearchCriteria fcsc );
    };

    interface FindCustomerDirectorInterface {
       oneway void requestBusinessRequest( in long eventId,
          in FindCustomerViewControllerInterface fc );
    };
```

• Table 5 - First successful interface

After using this interface successfully, we found that there were a number of problems with it. These are described below. After a number of iterations, the final interface (see Table 6) was completed.

```
// *********************************************************
// * IDL for Network Computing proof of concept *************
// *********************************************************
//
// file: POC11.IDL
// First created: 29 September 1997

Enum Events {
   FindCustomerByName,
   FindCustomerByAddress,
   FindCustomerByCity };

   struct Customer {
      string firstName;
      string lastName;
      string city;
   };

   typedef sequence<Customer> CustomerSequence;

   interface FindCustomerDirector {
```

```
                long requestBusinessRequest (
                    in Events EventId,
                    in string searchName,
                    inout CustomerSequence results );
        };
```

• Table 6 - Final interface

6.2.3 Compiling an IDL file

Once the interface has been finalised, and has been defined in an IDL file, it must then be compiled. The IDL compiler is different on each platform and implementation. On the Java platform, using OrbixWeb, the IDL compiler is simply called IDL.

When an IDL file is compiled, it produces two different types of files. The first type of file is what is known as a skeleton file. Skeleton files are essentially just what their name describes - skeletons which need bodies. For example, a method called populate was defined on the client, which was given a blank model object, and was expected to fill it with data. The IDL compiler took the IDL definition and produced a Java class that required implementing. I took this Java class, and filled in the relevant methods to produce the desired outcome of a fully populated model object.

6.2.4 Issues that were addressed

Deadlock

In the standard use of CORBA based applications, transactions are fulfilled in a synchronous nature. A client makes a call to the server. The server then performs its function, and returns a result to the client.

The model that we were required to work with was of an asynchronous nature. This means that when the transaction is completed the server must call the client. In our case the update method of the View Controller object on the client is called from the server.

Method calls on the server can be made asynchronous by using the *oneway* modifier. This can only be used when there are no values to be returned from the server. Callback type methods can be used to bypass this feature.

To ensure that deadlock did not occur, we used a combination of making asynchronous calls to the server, with a multi-threaded architecture on the client. This result was a client that could handle the requirements of the asynchronous architecture.

Too many calls to the server

Our initial implementation resulted in a large amount of network traffic. We found that this was due to the fact that CORBA does not use a "pass by value" mechanism, but instead just passes a reference to that object. Apparently, the "pass by value" mechanism may be defined in the future.

As an example, the following piece of code produced four separate network events for each iteration of the loop:

```
for ( int i; i<list.length; i++)

getList().getCustomer(i).getCustomerName().getLastName();
```

Each method resulted in a call to a remote object!

To reduce the amount of network activity, we had to redesign the interfaces. We found that objects containing only basic data types could be redefined as "structs". A struct is a data container that can hold different basic data types. When the client requests a struct, all of the data is copied across the network, rather than just a reference being passed.

We also found that a "sequence" data type could be used to hold multiple "structs". A sequence in similar in nature to the common array, except that it's length is dynamic in nature (it can be altered at runtime).

The Customer object was converted into a struct. The CustomerList then became a sequence of these Customer structs. By doing this, we were able to significantly reduce the number of calls across the network to a single call and response. Further access to the list would occur locally, since the data now resided on the client.

6.3 Java client

6.3.1 Requirements

The goal of the client application was to wrap the above functionality with a user-friendly interface. The interface should resemble the current C++ version of the "Find Customer" panel.

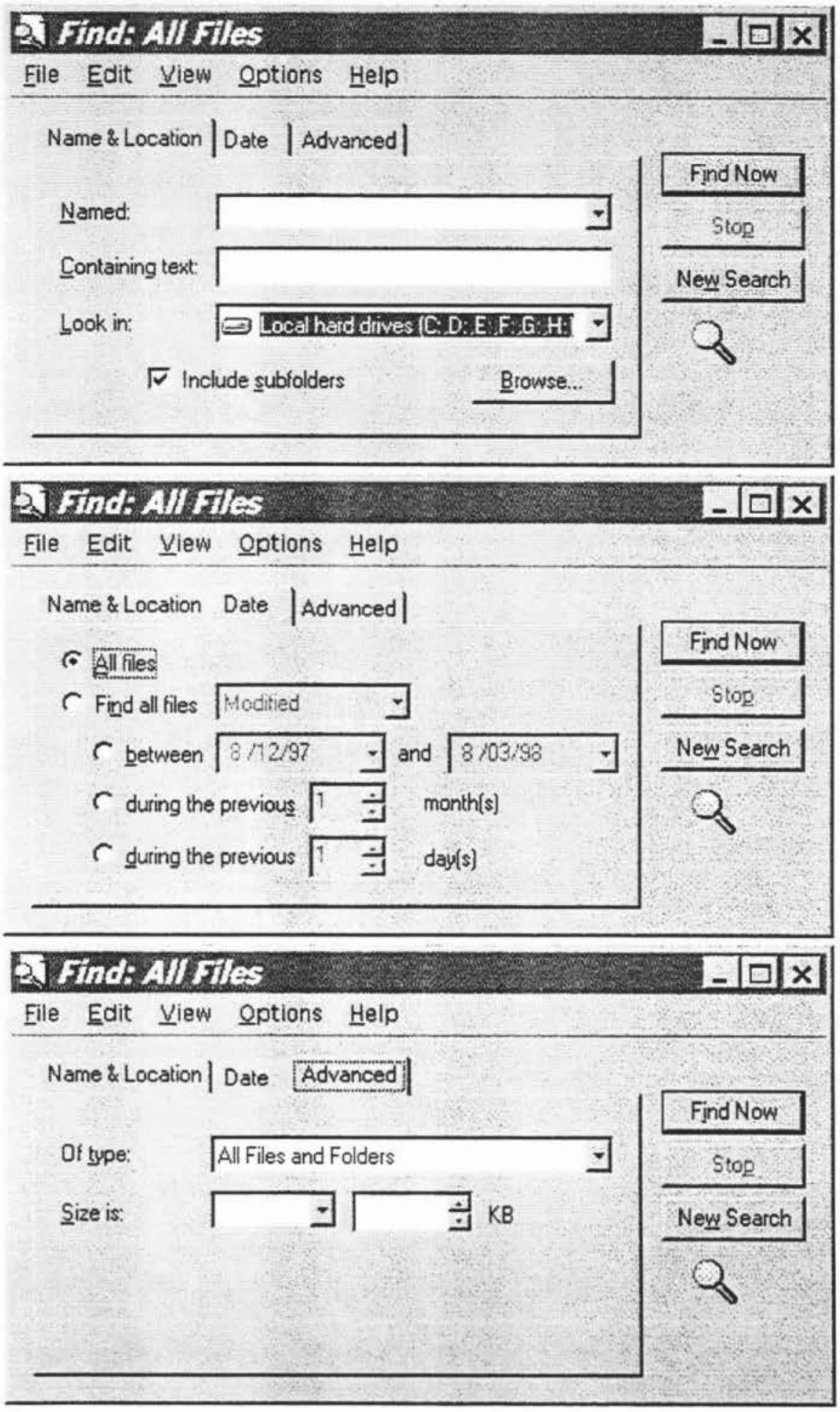
The components of the interface should be Java beans.

6.3.2 Implementation

Tabbed Notebook

When the interface was first designed, many of today's common user interface packages were not available. The components that we have to work with included the basic AWT objects, and a multicolumn listbox that came with the Visual Age for Java development environment.

One of problems was that there was no tabbed notebook UI component. A tabbed notebook such as the one shown in Figure 9, is a good example of what was needed for client of Find Customer. When a user clicks on one of the three tabs along the top, the screen below the tabs changes.


Find: All Files
File Edit View Options Help
Name & Location | Date | Advanced
Find Now
Stop
New Search
Browse...

Named:

Containing text:

Look in: Local hard drives (C: D: E: F: G: H:

☑ Include subfolders

Find: All Files
File Edit View Options Help
Name & Location Date | Advanced
Find Now
Stop
New Search

● All files

○ Find all files Modified

○ between 8/12/97 and 8/03/98

○ during the previous 1 month(s)

○ during the previous 1 day(s)

Find: All Files
File Edit View Options Help
Name & Location | Date Advanced
Find Now
Stop
New Search

Of type: All Files and Folders

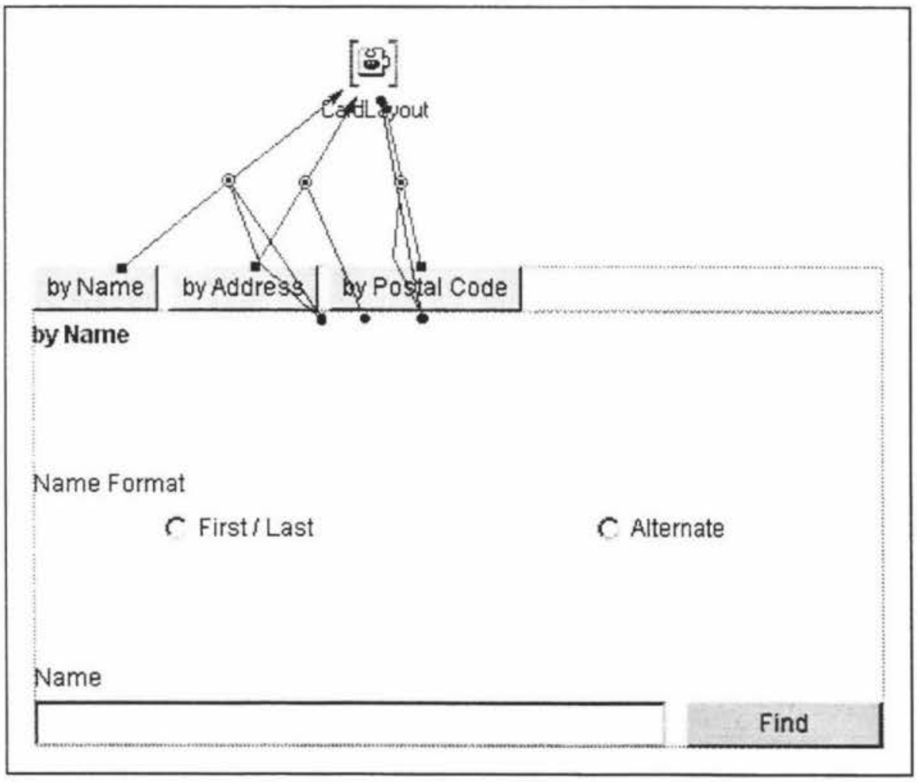
Size is: KB

- Figure 9 - Use of tabbed Notebooks

To realise the function of a tabbed notebook I used a combination of three Java AWT buttons along the top of the panel. Below these buttons, I added another panel with its layout manager set to CardLayout. The card layout manager allows the designer to place a number of different sets of objects on each "card". A current card can be selected, and the objects associated with it are then displayed. I used this functionality in combination with the buttons to simulate a tabbed notebook.

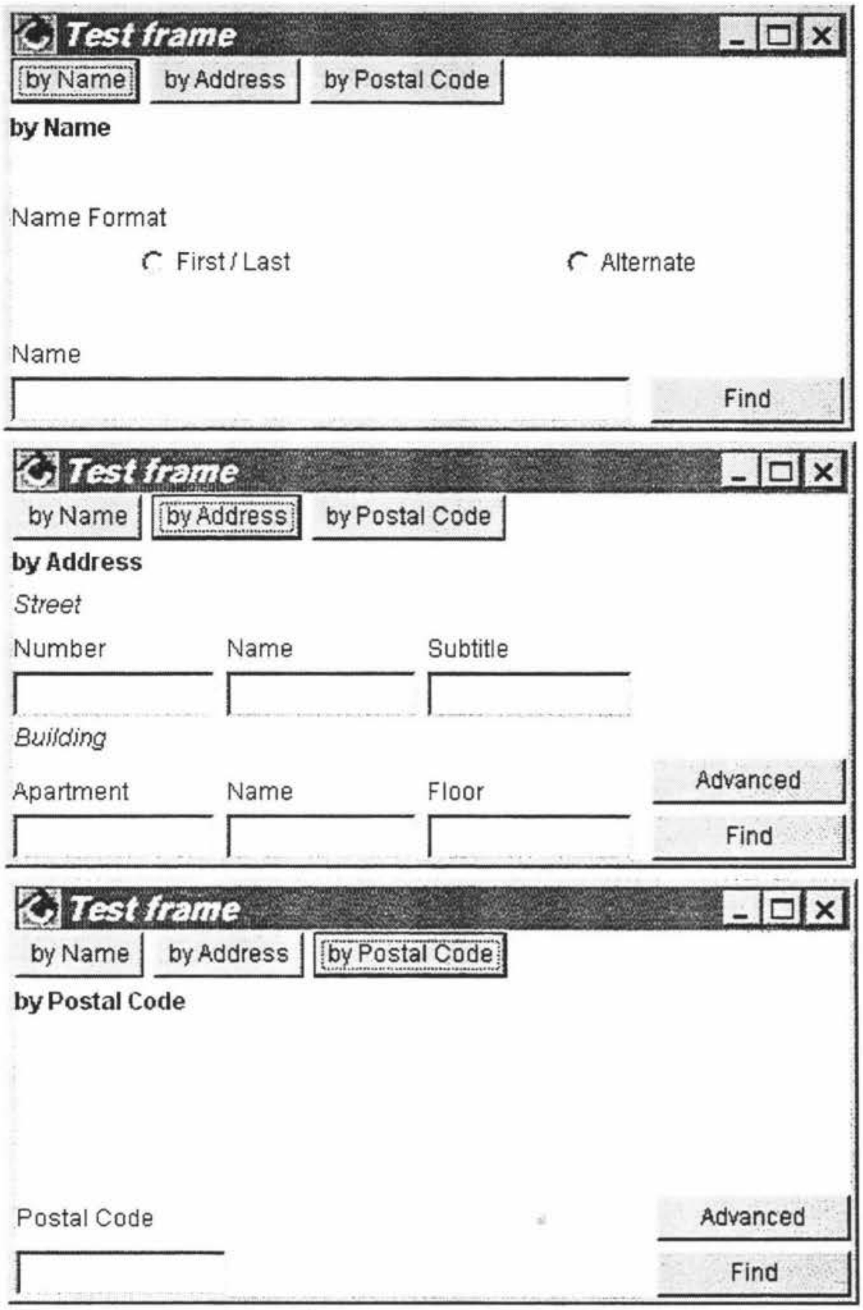
The implementation of the tabbed notebook functionality was completed with the help of the Visual Builder of Visual Age for Java. This tool allows you to create both user interfaces, as well as some of the functionality behind them. Figure 10 shows how Visual Programming is used – a card layout object is shown at the top of the diagram. This represents the layout property of the panel in the bottom of the diagram. When changes are made to this object, they are reflected in the actual layout object in the panel.

The buttons generate events when they are pressed. The arrows indicate the actions that are performed when these events occur. In the example in Figure 10, a method is called on the CardLayout object that changes the current "card" in the view. The result is that when one of the buttons is pressed, the display below it changes to reflect the new panel.



• Figure 10 - Visual programming

The above design was then compiled, and added to a simple test program. The results are shown below in Figure 11. Pressing the relevant button can access each of the different panels.



• Figure 11 - Implementation of Tabbed Notebook functionality

UI Components are Java Beans

All of the UI components were converted into Java Beans. This is a good practice to follow, since it promotes code reuse, and forces the developer to closely follow data abstraction rules. The Java Beans can be (and were) easily added to the component palette. They can now be easily added to other applications through a simple "drag and drop" process.

Multiple thread design

The NC client was designed in a way so that each of the major components would run in it's own thread. There were three main components that were run as separate threads. These were:

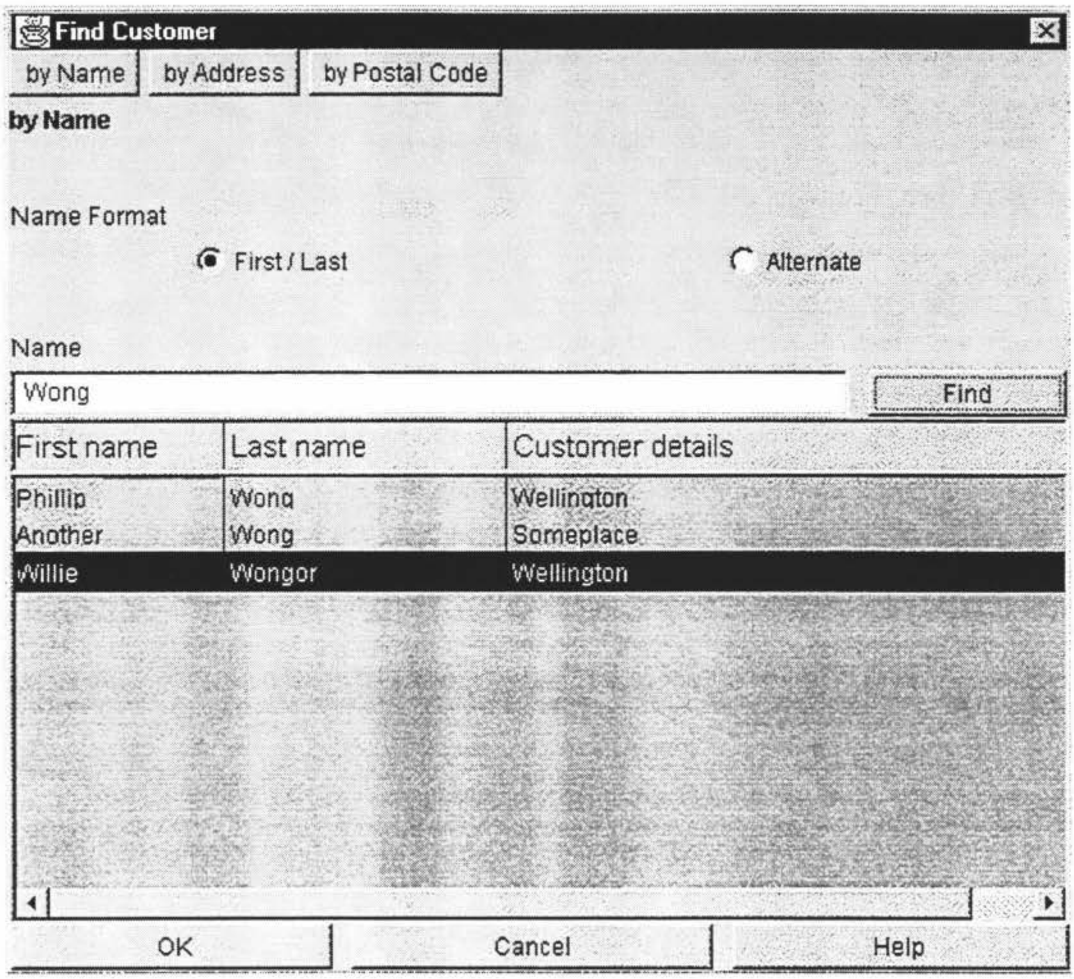
- The main application
- Each *FindCustomer* use case dialog box
- The CORBA communication monitor

This resulted in an application that would not suffer from deadlock.

6.3.3 Client Usage

When the program is first started, the user is presented with a plain screen with a number of menu options. Most of the options are "greyed out", and a designed to be implemented in further releases of the project. One of the options is to search for customers.

When the user selects the "Find Customer" option, a window is displayed. The user then enters the desired search criteria, and presses the find button. The results are then displayed in a list box below. See Figure 12 for an example.



by Name

Name Format

● First / Last ○ Alternate

Name: Wong

First name	Last name	Customer details
Phillip	Wong	Wellington
Another	Wong	Someplace
Willie	Wongor	Wellington

- Figure 12 - Find Customer usage

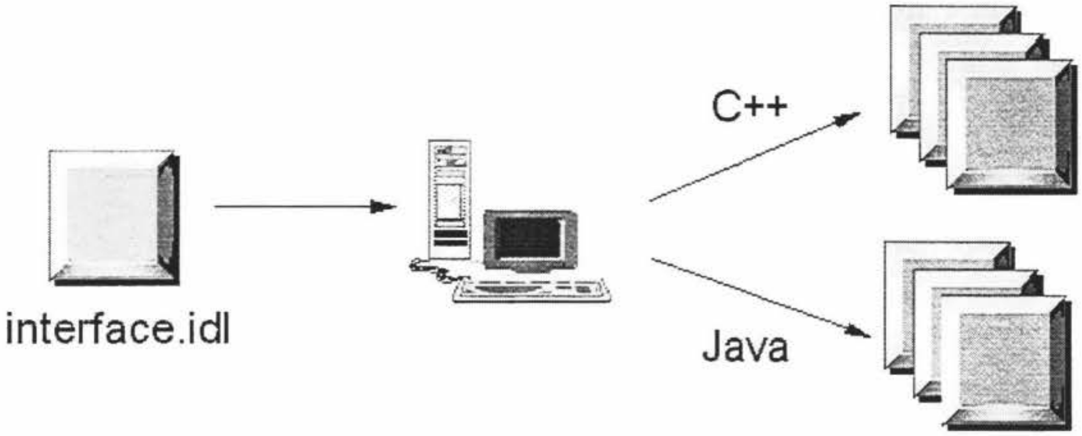
6.4 C++ server

The C++ server was based on the Voyager client. The Voyager client is a program that is used to perform everyday customer requests by a Customer Service Representative at a Telephone company. For example, service requests, billing information, etc.

One of the integral parts of the application is the ability to identify the "customer" who is calling. As has already been shown, this can be done in a number of ways – for example, "by name" or "by address". The functionality of searching for customers had already been implemented. What needed to be done was to integrate this current code with a CORBA interface.

6.4.1 CORBA Integration

The starting point to the CORBA integration was to take the IDL interfaces that were now already written, and to compile the C++ version of them. The result was similar to the Java version, except that we now had skeletons in the C++ language.



• Figure 13 - IDL compiling

These skeletons were then used to derive concrete classes from.

6.5 Performance

6.5.1 CORBA

We looked at how we could measure the performance of a CORBA implementation.

We found example software that would test a wide range of data types and allow selection of the buffer size. We had no problem making the software work and could quickly produce a range of statistics if these were required (testing many simultaneous clients would be more of a challenge).

6.5.2 Java

The apparent performance of the Java client was excellent. All operations other than large CORBA based requests resulted in sub-second performance. Many of the earlier performance issues that may have caused concern in the Java community appear to have been solved (or at least reduced) by the use of JIT (just in time) compilers.

Performance can only be expected to increase with the impending release of Sun's "High Performance Java." With the advent of specialised Java microchips, this too will be surpassed.

Massey University Computing Services

RL Ramsay

Name:	RL Ramsay
Printed from:	cc-stb2-36
Environment:	PRIVATE
Printed in:	UG Computing Laboratory
User:	student
Host:	cc-stb2-36
Date:	Tue Feb 29 12:41:17 2000

```
You have been allocated a resource budget of $44.41 for PRIVATE,
before this print job started you had $5.21 remaining.

                          RULES OF THE LAB
                          ----------------

* NO ID, No entry
* NO smoking eating or drinking
* NO playing games
* No dirty shoes or gumboots
* NOISE to be kept to a minimum
* NO copying of software
* NO visitors
* NO removing, tampering or otherwise interfering with computer equipment
* REPORT faults to supervisors
* PUT unwanted printing in recycle bins
```

6.6 Future enhancements

6.6.1 Split reassessment

The split between the client and server should be reassessed. One of the reasons the split was made between the controller and director classes was because of the possible performance difficulties of Java. It has been shown that the performance was adequate, and with further developments of Java performance in the future it may be possible to move a more significant of the overall code base onto the Java client.

6.6.2 Middle tier

In this project, there were actually four tiers – the client, the CORBA server, the real server, and the database. We believe that this is one too many. The CORBA server should be moved to the "real" server. If the split between client and server moves from the controller / director to the director / service, then that is even more reason to make the move. This is because the service objects currently reside on the "real" server.

Removing this unnecessary tier would have a number of tangible benefits:

6) Hardware costs would be reduced, as this extra tier would be eliminated. The work that it originally did would be moved to the Java client.

7) Performance would increase. By eliminating one of the tiers, we reduce the number of steps, and hence time that is required to run a transaction.

8) Increased stability. If the OS/2 tier were to fail, the whole system would currently halt. By removing of the tiers, we remove a potential point where the system could fall over.

9) Decreased risk – as was stated earlier in this chapter, there were very few CORBA solutions available for the OS/2 platform. If the company which manufactures the CORBA solution was to collapse, then the project would be at risk as there would be no further development or support of the CORBA line. The server platform that is AIX currently has many different solutions available for it.

6.6.3 Data structures

The data structures that were used in this project were all based upon the *struct* data type. This is because we had difficulty in getting more complex data structures to be passed across the interface between client and server. Further investigation needs to be done to determine why these difficulties were experienced, and how a work-around could be achieved.

6.6.4 CORBA services

CORBA has a number of services than can assist this project. Further research should be made into how services such as "Naming" and "Security" can be used. The security service could be used to allow this project to be extended to the Internet to allow online transactions.

6.7 Conclusions

It has been shown that CORBA can be used to bind two different platforms and two different programming languages together. The Java language and the C++ language match each other well.

Modern Java technologies such as Java Beans and the use of the JDK1.1 event model were demonstrated.

Performance was shown to be excellent.

A number of future topics of research have been given. A reassessment of the split position between client and server should be made. Removal of the middle tier may lead to many improvements. Assessment and extension of current data structures may lead to other benefits. Finally, research into other CORBA services should be conducted.

All of the aims of this project have been satisfied.

7. Conclusions

It has been shown through use of two practical projects that the Internet can be used to add value to current data sets and existing applications.

7.1 Using a current data set

The first project (Toll Estimator) involved simulating the use of records from a telephone company. Rating data that resided in a DB2 database was used to provide an application that allowed a customer to find out the cost of a potential phone call. Internet technologies such as Java and its communications component called Remote Method Invocation (RMI) were used to create this application. This "applet" which can run inside a standard web browser was easy to run and use. All the customer had to do was go to a certain web page, and the program would start automatically.

7.2 Using a current application

The second project (Find Customer) involved taking an existing application, and adding a Java based front-end to part of its functionality. The application used was C++ based, and was part of an application used to answer customer's queries. The functionality that was extended was a subsystem that provided search capabilities for finding customers matching certain criteria.

This existing functionality was wrapped with a CORBA interface, and made available with an Object Request Broker. A Java based client was then written which made a connection to the defined CORBA interface, made a call to that subsystem, passing a defined search criteria. The results were then received back, and displayed in the Java application.

7.3 The use of Internet based technologies

Internet based technologies were used in the two projects mentioned above. One of the reasons for the success of these projects was because of the use of industry based standards. These standards allow different applications from different manufacturers to operate with each other.

It was shown how a DB2 database developed by IBM was used in conjunction with a Java (Sun Microsystems) program which ran inside a web browser from Netscape Communications. It was also shown how an existing C++ based application developed by IBM New Zealand could be linked using a CORBA system developed by Orbix, to a Java (Sun Microsystems) based application.

By using these common standards, programs can interoperate without too much difficulty. The Internet has further strengthened the use of these standards with the availability of web browsers and their embedded Java Virtual Machines.

With the increasing popularity of the Internet, and the growth of Internet based applications, it would seem likely that there will be more opportunity for existing applications and data sets to be extended to either the Internet itself, or internal corporate use.

8. Appendix A – Toll Estimator documentation

8.1 Package COM.ibm.voyager

Interface Index

- Callable
- Database

Class Index

- CallCost
- CallInfo
- Message
- Messenger
- TollEstimator
- TollFrame

8.1.1 Interface COM.ibm.voyager.Callable

public interface Callable

The Callable interface provides a mechanism for making callbacks from a messenger that executes in a separate thread.

Version:

1.1.1

Author:

Nigel Ramsay

See Also:

Message, Messenger

Method Index

setCallCost(Message)

Called by a messenger to respond to a request for the cost of a call The CallCost object that is contained in the contents of the message is filled with the details about the cost of the call.

setLocations(Message)

Called by a messenger to respond to a request for the list of locations.

setMessenger(Messenger)

Called by a messenger to respond to inform the implemented version of this abstract class of the messenging object that it will use for making requests for information.

Methods

setCallCost

```
public abstract void setCallCost(Message arg)
```

Called by a messenger to respond to a request for the cost of a call The CallCost object that is contained in the contents of the message is filled with the details about the cost of the call. The implemented version of this abstract method is called in response to a request made to the messenger by calling myMessenger.requestCallCost(this, myCallInfo);

Parameters:

arg - Message containing CallCost object (and possible error strings)

See Also:

Messenger, CallCost

setLocations

`public abstract void setLocations(Message arg)`

Called by a messenger to respond to a request for the list of locations. The Vector object which is contained in the contents of the message is filled with strings of each of the different locations. If any errors have occured then the message will contain them. The implemented version of this abstract method is called in response to a request made to the messenger by calling myMessenger.requestLocations(this);

Parameters:

arg - Message whose contents are a vector

See Also:

Messenger, Vector

setMessenger

`public abstract void setMessenger(Messenger arg)`

Called by a messenger to respond to inform the implemented version of this abstract class of the messenging object that it will use for making requests for information. The creator of the implemented version of this abstract class must be aware that they can not make any calls of the messenger until this method has been called. The checking could be done as follows:

```
while(messenger==null){
     try {
        Thread.sleep(100);
     } catch (Exception e) {
        System.out.println("error message goes here");
     }
 }
```

Parameters:

arg - The messenger object that will respond to request for information

8.1.2 Interface COM.ibm.voyager.Database

public interface Database

extends Remote

The Database remote interface provides a mechanism for making specific calls to a database.

Version:

1.1.1

Author:

Nigel Ramsay

See Also:

DatabaseImpl

Method Index

getCallCost(CallInfo)

This abstract method makes a query to a database for the cost of an intended telephone call.

getLocations()

This abstract method makes a query to a database for a list of the different locations from where users can make telephone calls to or from.

Methods

getCallCost

```
public abstract Message getCallCost(CallInfo arg)
  throws RemoteException
```

This abstract method makes a query to a database for the cost of an intended telephone call. The methods extracts the data from the CallInfo object, which it uses for the database query. The results are then packaged up into a Message object. The results stored inside the Message object are packaged inside a CallCost object. If any errors occur while attempting to access to retrieve data from the database, these will also be stored inside the Message object.

See Also:

Message, DatabaseImpl, CallCost

getLocations

```
public abstract Message getLocations() throws
   RemoteException
```

This abstract method makes a query to a database for a list of the different locations from where users can make telephone calls to or from. The results are packaged up into a Message object. The results stored inside the Message object are packaged inside a Vector which contains a number of strings. If any errors occur while attempting to access to retrieve data from the database, these will also be stored inside the Message object.

See Also:

Message, DatabaseImpl, Vector

8.1.3 Class COM.ibm.voyager.CallCost

```
java.lang.Object
   |
   +----COM.ibm.voyager.CallCost
```

public class CallCost

extends Object

implements Serializable

The CallCost class is used for holding call cost data

Version:

1.1.1

Author:

Nigel Ramsay

See Also:

CallCost, CallInfo

Constructor Index

CallCost()

Constructs a CallCost object with variables initialised to 0

CallCost(float, float)

Constructs a new CallCost object with variables set to the supplied parameters.

Method Index

getBaseRate()

Returns the base rate of the call.

getTotalCost()

Returns the total cost of the call.

setBaseRate(float)

Sets the base rate of the call.

setTotalCost(float)

Sets the total cost of the call.

Constructors

CallCost

public CallCost()

Constructs a CallCost object with variables initialised to 0

CallCost

```
public CallCost(float newBaseRate, float newTotalCost)
```

Constructs a new CallCost object with variables set to the supplied parameters.

Parameters:

newBaseRate - The base rate of the call. Units = dollars per minute.

newTotalCost - The total cost of a call. Units = dollars.

Methods

getBaseRate

```
public float getBaseRate()
```

Returns the base rate of the call.

Returns:

the base rate.

getTotalCost

```
public float getTotalCost()
```

Returns the total cost of the call.

Returns:

the total cost.

setBaseRate

```
public void setBaseRate(float arg)
```

Sets the base rate of the call.

Parameters:

arg - The base rate.

setTotalCost

```
public void setTotalCost(float arg)
```

Sets the total cost of the call.

Parameters:

arg - The total cost

8.1.4 Class COM.ibm.voyager.CallInfo

```
java.lang.Object
   |
   +----COM.ibm.voyager.CallInfo
```

public class CallInfo

extends Object

implements Serializable

The CallInfo class is used for holding the details of a proposed telephone call

Version:

1.1.1

Author:

Nigel Ramsay

See Also:

CallCost

Constructor Index

CallInfo()

Constructs the CallInfo object.

CallInfo(String, String, String, int, int, String, int)

Constructs the CallInfo object with variables set to the supplied parameters.

Method Index

getAmPm()

Returns a string indicating if the indended call is either before or after noon

getDay()

Returns a string containg the day name of an intended call

getFrom()

Returns a string containing the location from which an intended call is to be made

getHour()

Returns an integer containg the hour of an intended call.

getLength()

Returns an integer containg the length of an intended call.

getMinute()

Returns an integer containg the minute of an intended call.

getTo()

Returns a string containing the location to which an intended call is to be made

setAmPm(String)

Sets the call time as either AM or PM to indicate if the indended call is either before or after noon

setDay(String)

Sets the day name of an intended call

setFrom(String)

Sets the location from which an intended call is to be made

setHour(int)

Sets the hour of an intended call.

setLength(int)

Sets the hour of an intended call.

setMinute(int)

Sets the minute of an intended call.

setTo(String)

Sets the location to which an intended call is to be made

Constructors

CallInfo

```
public CallInfo()
```

Constructs the CallInfo object. The variables are initialised to

```
From = "";
To = "";
Day = "";
Hour = 12;
Minute = 0;
Length = 0;
AmPm = "am";
```

CallInfo

```
public CallInfo(String newFrom,
                String newTo,
                String newDay,
                int newHour,
                int newMinute,
                String newAmPm,
                int newLength)
```

Constructs the CallInfo object with variables set to the supplied parameters.

Parameters:

newFrom - String containing the From location.

newTo - String containing the To location.

newDay - String containing name of the day on which the intended call is to be made.

newHour - Hour on which the intended call is to be made, where the value is between

1 and 12

newMinute - Minute on which the intended call is to be made

newAmPm - String containing in lowercase either

`am`
or

`pm`
newLength - Length of the call in minutes

Methods

getAmPm

```
public String getAmPm()
```

Returns a string indicating if the indended call is either before or after noon

Returns:

Either "am" or "pm"

getDay

```
public String getDay()
```

Returns a string containg the day name of an intended call

Returns:

Day name. eg: "Monday"

getFrom

```
public String getFrom()
```

Returns a string containing the location from which an intended call is to be made

Returns:

From location name. eg: "Petone"

getHour

`public int getHour()`

Returns an integer containg the hour of an intended call.

Returns:

Hour. The value will be between 1 and 12. eg: 10

getLength

`public int getLength()`

Returns an integer containg the length of an intended call. The value will be between 0 and java.Integer.MAX_VALUE

Returns:

Call length. eg: 120

getMinute

`public int getMinute()`

Returns an integer containg the minute of an intended call. The value will be between 0 and 59.

Returns:

Minute. eg: 45

getTo

`public String getTo()`

Returns a string containing the location to which an intended call is to be made

Returns:

To location name. eg: "Petone"

setAmPm

`public void setAmPm(String arg)`

Sets the call time as either AM or PM to indicate if the indended call is either before or after noon

Parameters:

arg - Either am or pm

setDay

```
public void setDay(String arg)
```

Sets the day name of an intended call

Parameters:

arg - Day name. eg: "Monday"

setFrom

```
public void setFrom(String arg)
```

Sets the location from which an intended call is to be made

Parameters:

arg - Location name. eg: "Petone"

setHour

```
public void setHour(int arg)
```

Sets the hour of an intended call.

Parameters:

arg - Hour. The value must be between 1 and 12. eg: 10

setLength

```
public void setLength(int arg)
```

Sets the hour of an intended call.

Parameters:

arg - Call length. The value must be between 0 and java.lang.Integer.MAX_VALUE.

eg: 10

setMinute

```
public void setMinute(int arg)
```

Sets the minute of an intended call.

Parameters:

arg - Minute. The value must be between 0 and 59. eg: 10

setTo

```
public void setTo(String arg)
```

Sets the location to which an intended call is to be made

Parameters:

arg - Location name. eg: "Petone"

8.1.5 Class COM.ibm.voyager.Message

```
java.lang.Object
   |
   +----COM.ibm.voyager.Message
```

public class Message

extends Object

implements Serializable

The Message class provides a container for holding an object for sending between both threads and remote objects. What makes this class special is that it also provides a way in which error messages may be passed which having to cause exceptions to occur. This class was created to solve the problem of indicating the presence of an error to a callback function. Each message may contain one object and any number of error messages.

Version:

1.1.1

Author:

Nigel Ramsay

Constructor Index

Message()

Method Index

addError(String)

The addError method takes a string and adds it to the list of strings that may have already been added.

getContents()

The getContents method returns contents of the Message

getError()

The getError method returns an error message.

isContents()

The isContents method tests to see if an object has been added to this message using the setContents method.

isError()

The isError method tests to see if an error has been added to this message using the addError method.

setContents(Object)

The setContents method takes any object and adds it to the message.

Constructors

Message

```
public Message()
```

Methods

addError

```
public void addError(String arg)
```

The addError method takes a string and adds it to the list of strings that may have already been added.

Parameters:

arg - The error message

getContents

```
public Object getContents()
```

The getContents method returns contents of the Message

Returns:

The object that was stored with the setContents method. If the setContents method was not called, then this method will return null

See Also:

setContents

getError

```
public String getError()
```

The getError method returns an error message. Example code:

```
while(arg.isError()){
    statusString = arg.getError();
    statusString= statusString.concat(arg.getError());
    statusString = statusString.concat("; ");
}
```

Returns:

A single error message. The act of getting this error will also remove it from the list. You can also check to see if an error exists by calling the isError method. Errors are removed from the Message in the opposite order to which they were added (ie: it is an error stack). If no error exists, then this method returns an empty string.

See Also:

setContents

isContents

```
public boolean isContents()
```

The isContents method tests to see if an object has been added to this message using the setContents method.

Returns:

True if this message contains an object. Otherwise, false.

See Also:

setContents

isError

```
public boolean isError()
```

The isError method tests to see if an error has been added to this message using the addError method.

Returns:

True if this message contains at least one error. Otherwise, false.

See Also:

addError

setContents

```
public void setContents(Object arg)
```

The setContents method takes any object and adds it to the message.

Parameters:

arg - Any object

8.1.6 Class COM.ibm.voyager.Messenger

```
java.lang.Object
   |
   +----COM.ibm.voyager.Messenger
```

requestCallCost(Callable, CallInfo)

This method is called by external classes to make a request for the cost of a specific call.

requestLocations(Callable)

This method is called by external classes to make a request for the list of locations.

run()

This method monitors for calls to the request type methods, and if any calls are made, it in turns makes a call to the corresponding execute type methods.

start()

This method starts the Messenger thread.

stop()

This method stops the Messenger thread.

Constructors

Messenger

```
public Messenger(String arg)
```

Contructs a new Messenger object. Because the messenger is run as a seperate thread, the start method must be called before any transactions will be processed.

Parameters:

arg - The URL of the remote server. The example:

```
Messenger messenger = new
  Messenger("rmi://java.ibm.com:2001/myMessenger");
```

See Also:

start

Methods

requestCallCost

```
public void requestCallCost(Callable caller, CallInfo
  info)
```

This method is called by external classes to make a request for the cost of a specific call. This method notes the request, and then returns control to the calling class. After the request has been processed by this thread, a call is made to the setCallCost() method of the calling class with the results of the request.

Parameters:

caller - A reference to the calling class

info - A class containing all the info about the inteded call

See Also:

setCallCost

requestLocations

```
public void requestLocations(Callable arg)
```

This method is called by external classes to make a request for the list of locations. This method notes the request, and then returns control to the calling class. After the request has been processed by this thread, a call is made to the setLocations() method of the calling class with the results of the request.

Parameters:

caller - A reference to the calling class

See Also:

setLocations

run

```
public void run()
```

This method monitors for calls to the request type methods, and if any calls are made, it in turns makes a call to the corresponding execute type methods. This method also makes the connection to the RMI server.

See Also:

executeCallCostRequest, executeLocationsRequest, requestCallCost, requestLocations

start

```
public void start()
```

This method starts the Messenger thread.

stop

```
public void stop()
```

This method stops the Messenger thread.

8.1.7 Class COM.ibm.voyager.TollEstimator

```
java.lang.Object
   |
   +----java.awt.Component
           |
           +----java.awt.Container
                   |
                   +----java.awt.Panel
                           |
                           +----java.applet.Applet
                                   |
                                   +----
  COM.ibm.voyager.TollEstimator
```

public class TollEstimator

extends Applet

implements Runnable

This class is an applet that provides toll call estimation facilities to web users.

Version:

1.1.1

Author:

Nigel Ramsay

Constructor Index

TollEstimator()

Method Index

destroy()

Handle the Applet destroy method.

getAppletInfo()

Information about this applet.

getParameterInfo()

This method returns information about the parameters that need to be supplied to make the applet work correctly.

init()

This method initialises the applet, and instantiates the Messenger and TollFrame classes

paint(Graphics)

This method currently does nothing.

run()

This method does nothing but continuely look

start()

This method starts up the TollFrame and Messenger classes

stop()

This method stops the TollFrame and Messenger classes.

Constructors

TollEstimator

```
public TollEstimator()
```

Methods

destroy

```
public void destroy()
```

Handle the Applet destroy method.

Overrides:

destroy in class Applet

getAppletInfo

```
public String getAppletInfo()
```

Information about this applet.

Returns:

The title of the applet

Overrides:

getAppletInfo in class Applet

```
public void start()
```

This method starts up the TollFrame and Messenger classes

Overrides:

start in class Applet

stop

```
public void stop()
```

This method stops the TollFrame and Messenger classes.

Overrides:

stop in class Applet

8.1.8 Class COM.ibm.voyager.TollFrame

```
java.lang.Object
   |
   +----java.awt.Component
           |
           +----java.awt.Container
                   |
                   +----java.awt.Window
                           |
                           +----java.awt.Frame
                                   |
                                   +----
  COM.ibm.voyager.TollFrame
```

public class TollFrame

extends Frame

implements Callable, ActionListener, Runnable

This class provides all of the visual needs of the TollEstimator applet. Most of the visual components were developed using the Visual Age / Java visual builder. This class implements the Callable interface which lets it make requests of Messenger class for information from the database. This class also implements the Runnable interface, and is threaded.

Constructor Index

Constructor

Returns:

java.awt.Frame

Methods

actionPerformed

```
public void actionPerformed(ActionEvent e)
```

Method to handle events for the ActionListener interface.

Parameters:

e - java.awt.event.ActionEvent

handleEvent

```
public boolean handleEvent(Event evt)
```

Method to handle old AWT events

Parameters:

evt - java.awt.Event

Returns:

boolean

Overrides:

handleEvent in class Component

run

```
public void run()
```

TollFrame.run method comment.

setCallCost

```
public void setCallCost(Message arg)
```

TollFrame.setCallCost method comment.

9. Appendix B – Find Customer documentation overview

9.1 Package gui12

9.1.1 Interfaces

- MainViewListener

9.1.2 Classes

- EventProcessor
- MainController
- MainView
- MainViewListenerEventMulticaster
- Voyager

9.2 Package fc12

9.2.1 Interfacess

- ByAddressPanelListener
- ByNamePanelListener
- ByPostalCodePanelListener
- FindCustomerPanelListener

9.2.2 Classes

- ByAddressPanel
- ByAddressPanelListenerEventMulticaster
- ByNamePanel
- ByNamePanelListenerEventMulticaster

- ByPostalCodePanel
- ByPostalCodePanelListenerEventMulticaster
- FindCustomerPanel
- FindCustomerPanelListenerEventMulticaster
- FindCustomerView
- FindCustomerViewController
- SearchPanel

9.3 Package poc12

9.3.1 Interfaces

_FindCustomerDirectorRef

9.3.2 Classes

Customer

CustomerSequence

Events

FindCustomerDirector

_FindCustomerDirectorHolder

_sequence_Customer

10. References

1. Terry Sullivan, "As Simple as Possible", *The Usable Web*, April 1997 (http://www.pantos.org/atw/35504.html).
2. T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.0", *RFC 1945*, May 1996 (http://www.w3.org/pub/WWW/Protocols/rfc1945/rfc1945).
3. Netscape Communications Corporation, "PERSISTENT CLIENT STATE HTTP COOKIES", June 1997 (http://home.netscape.com/newsref/std/cookie_spec.html)
4. David Whalen, "The Unofficial Netscape Cookie FAQ", *Cookie Central*, May 1997 (http://www.cookiecentral.com/unofficial_cookie_faq.htm).
5. Arman Danesh, Teach Yourself JavaScript in a week, Sams Publishing, 1996.
6. National Center for Supercomputing Applications, "The Common Gateway Interface", December 1995 (http://hoohoo.ncsa.uiuc.edu/cgi/).
7. Kaveh Basiri, "Programming with the NSAPI", *Proc. of Netscape Developers' Conference*, March 1996 (http://home.netscape.com/misc/developer/conference/proceedings/s5/index.html).
8. Microsoft Corporation, "Internet Server API Documentation", July 1996 (http://www.microsoft.com/win32dev/apiext/isalegal.htm).
9. Netscape Communications Corporation, "Enterprise Server Programmer's Guide", June 1997 (http://developer.netscape.com/library/documentation /enterprise/nt/index.html, http://developer.netscape.com/library /documentation/enterprise/unix/index.html).
10. Larry Wall, "What is Perl?", June 1996 (http://www.perl.com/perl/info /synopsis.html).
11. Yahoo, "SQL", June 1997 (http://www.yahoo.com/Computers_and_Internet/Programming_Languages/SQL/).
12. Sun Microsystems, Inc, "JDBC Guide: Getting Started", 1997 (ftp://ftp.javasoft.com/docs/jdk1.1/jdbc.pdf).
13. Object Management Group, "What is CORBA?", 1999 (http://www.omg.com/corba/whatiscorba.html)